МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский государственный университет экономики, статистики и информатики

Филиппов В.А.

УЧЕБНОЕ ПОСОБИЕ ПО ДИСЦИПЛИНЕ «API и COM технологии Windows»

Москва 2004

Филиппов В.А., Учебное пособие по дисциплине «API и COM технологии Windows» - М. Московский государственный университет экономики статистики и информатики. 2004. – 249 с.

© Филиппов В.А., 2004
© Московский государственный университет экономики статистики и информатики, 2004

Содержание

1. Интерфейс прикладного программирования Win32 4
   1.1. Основной код API Win32 4
   1.2. 32- и 16-разрядные компоненты 5
2. Введение в COM. Компоненты. 6
3. Интерфейс 15
4. QueryInterface 28
5. Подсчет ссылок 45
6. Динамическая компоновка 61
7. HRESULT, GUID, Реестр и другие детали 72
8. Фабрика класса 92
9. Повторная применимость компонентов: включение и агрегирование 112
10. Будем проще 148
11. Серверы в EXE 175
12. Диспетчерские интерфейсы и автоматизация 199
13. Многопоточность 224
14. Заключение 248
15. Список литературы 249

1. Интерфейс прикладного программирования Win32

Интерфейс прикладного программирования (Application Programming Interface, API) Win32 обеспечивает приложениям доступ ко всему спектру функций операционных систем семейства Windows. Функции, сообщения и структуры Win32 формируют последовательный и единообразный API для Windows 95 и Windows NT. Средства API Win32 позволяют разрабатывать приложения, успешно работающие на всех платформах и в то же время при надобности использующие уникальные особенности любой из них.
Многие функции API интегрированы в состав таких программ, как Visual Basic. Например, функцию API Win32 **MessageBox** можно вызвать непосредственно или через функцию Visual Basic **MsgBox.** Средствами Visual Basic обычно пользоваться легче, однако во многих случаях разработчики найдут непосредственное применение и самим функциям API Win32.

1.1. Основной код API Win32

Базовый код API Win32 содержится в трех библиотеках динамической загрузки (Dynamic Link Library, DLL): USER32, GDI32 и KERNEL32.

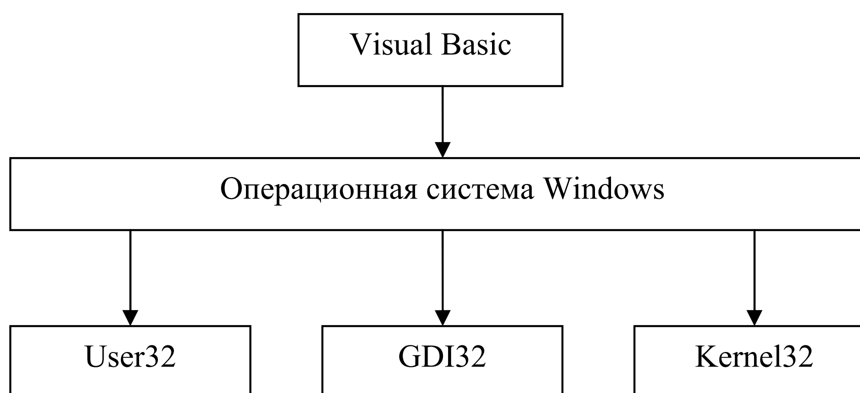


Рис. 1.1 *Библиотеки API Win32*

USER32

User32.dll и User.exe создают и контролируют окна на экране, выполняя все запросы по созданию, перемещению, изменению размеров и уничтожению окон. **User.exe,** кроме того, обрабатывает запросы, относящиеся к значкам и другим элементам интерфейса пользователя, а также переадресует события, порожденные различными устройствами ввода, соответствующим приложениям.

GDI32

Gdi32.dll и Gdi.exe контролируют интерфейс графических устройств (Graphics Device Interface, GDI). GDI выполняет графические операции при создании изображения на системном дисплее и других устройствах, включая:

- вывод на экран;
- вывод на принтер;
- включение/отключение пикселов.

KERNEL32

Kernel32.dll выполняет базовые функции операционной системы, в том числе:
управление памятью;

- файловый ввод/вывод;
- загрузку программы;
- выполнение программы.

Примечание При объявлении функций API в Visual Basic 32-разрядные функции чувствительны к регистру символов, а эквивалентные 16-разрядные функции — нет. Это необходимо иметь в виду при преобразовании 16-разрядных приложений в 32-разрядные.

1.2. 32- и 16-разрядные компоненты

В Windows 95 включены 16-разрядные версии User, GDI и Kernel. Комбинация 16-разрядного и 32-разрядного кода позволяет сохранить совместимость с суще ствующими приложениями и драйверами и одновременно увеличить производительность системы по сравнению с Windows 3.1. Windows 95 использует 32-разрядный код везде, где это увеличивает производительность не в ущерб совместимости. Для включения в Windows 95 16-разрядных компонентов есть три основные причины:

- код для 16-разрядных систем обеспечивает обратную совместимость с при ложениями и драйверами, разработанными для Windows 3.1;
- в некоторых случаях 16-разрядный код выполняется быстрее, чем аналогичный 32-разрядный;
- 32-разрядный код требует больше памяти, чем эквивалентный 16-разрядный.

Одна из основных задач Windows 95 — эффективная работа на компьютере с ограниченным объемом ОЗУ, и применение 16-разрядного кода «способствует» решению этой задачи.

Подсистемы ввода/вывода и драйверы устройств, включая сетевые и файловые системы, являются полностью 32-разрядными, как и все компоненты управления памятью и планирования. Часто возникающая при этом проблема вызова 32-разрядной функции из 16-разрядного приложения (или наоборот) решается при помощи *шлюзования.*

Шлюзование

Эта операция выполняется, когда операционная система преобразует вызов 16-разрядной функции в вызов 32-разрядной. Процессы Windows 95 и Windows NT не могут содержать одновременно и 16-разрядный, и 32-разрядный код. Шлюз позволяет коду с одной стороны границы вызывать код с другой ее стороны. Каждая платформа использует один или несколько механизмов шлюзования:

- механизм *базовых шлюзов* позволяет 16-разрядному Windows-приложению в системе под управлением Windows 95 и Windows NT загрузить и вызвать 32-разрядную библиотеку;
- с помощью механизма *плоских шлюзов,* реализованного только в Windows NT, Win32-приложение загружает и вызывает 16-разрядную библиотеку и наоборот.

2. Введение в COM. Компоненты.

Обычно приложение состоит из одного монолитного двоичного файла. После того, как приложение сгенерировано компилятором, он остается неизменным — пока не будет скомпилирована и поставлена пользователю новая версия. Чтобы учесть изменения в операционных системах, аппаратуре и желаниях пользователей, приходится ждать перекомпиляции. Приложение застывает, подобно скале посреди реки перемен.
И по мере того, как вся индустрия программирования стремительно уходит все дальше в будущее, оно стареет — и устаревает.

При современных темпах развития индустрии программирования приложениям нельзя оставаться застывшими. Разработчики должны найти способ вдохнуть новую жизнь в программы, которые уже поставлены пользователям. Решение состоит в том, чтобы разбить монолитное приложение на отдельные части, или компоненты (рис. 2.1).

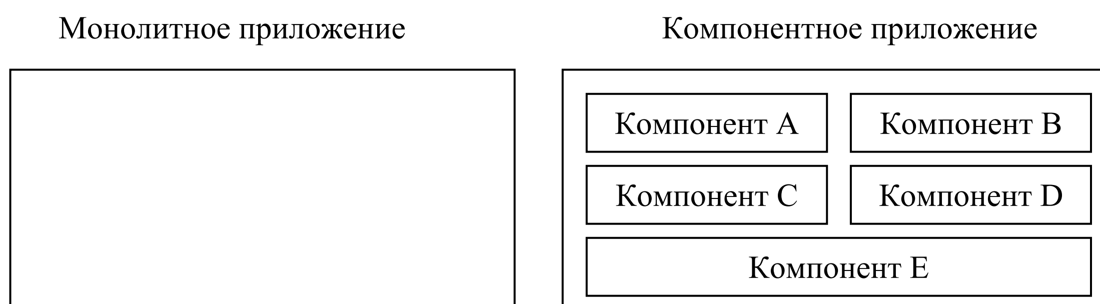


Рис. 2.1 *Разбиение монолитного приложения (слева) на компоненты (справа) облегчает адаптацию*

По мере развития технологии компоненты, составляющие приложение, могут заменяться новыми (рис. 2.2). Приложение более не является статичным, обреченным устареть еще до выхода в свет. Вместо этого оно постепенно эволюционирует с заменой старых компонентов новыми. Из существующих компонентов легко создать и абсолютно новые приложения.

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

Рис. 2.2 *Замена компонента D на новую, улучшенную версию*

Для того, чтобы разбить монолитное приложение на компоненты, необходим мощный инструмент. Инструмент, который мы будем использовать, называется *COM*. COM — *модель компонентных объектов* (*Component Object Model*) — это спецификация метода создания компонентов и построения из них приложений. Более восьми лет назад COM была разработана в Microsoft, чтобы сделать программный продукты фирмы более гибкими, динамичными и настраиваемыми. Практически все продаваемые сегодня приложения Microsoft используют COM. Технология ActiveX этой фирмы построена на основе компонентов COM.

Преимущества использования компонентов

Мы уже упоминали одно из преимуществ компонентных архитектур — способность приложения эволюционировать с течением времени. Кроме удобства и гибкости при модернизации существующих приложений, создание программ из компонентов имеет другие достоинства. Они связаны с адаптацией приложений к нуждам пользователя, библиотеками компонентов и распределенными компонентами.

Адаптация приложений

Пользователи часто хотят подстроить приложения к своим нуждам, точно так же, как мы подбираем домашнюю обстановку к своим вкусам. Конечные пользователи предпочитают, чтобы приложение работало так, как они привыкли. Программистам в корпорациях нужны адаптируемые приложения, чтобы создавать специализированные решенияна основе готовых продуктов. Компонентные архитектуры хорошо приспособлены для адаптации, так как любой компонент можно заменить другим, более соответствующим потребностям пользователя.

Предположим, что у нас есть компоненты на основе графических редакторов *Adobe Photoshop* и *Corel Draw*. Как видно из рис. 2.3, пользователь 1 может настроить приложение на использование *Adobe Photoshop*, а 2 — предпочесть *Corel Draw*. Приложения можно легко настраивать, добавляя новые компоненты или заменяя имеющиеся.

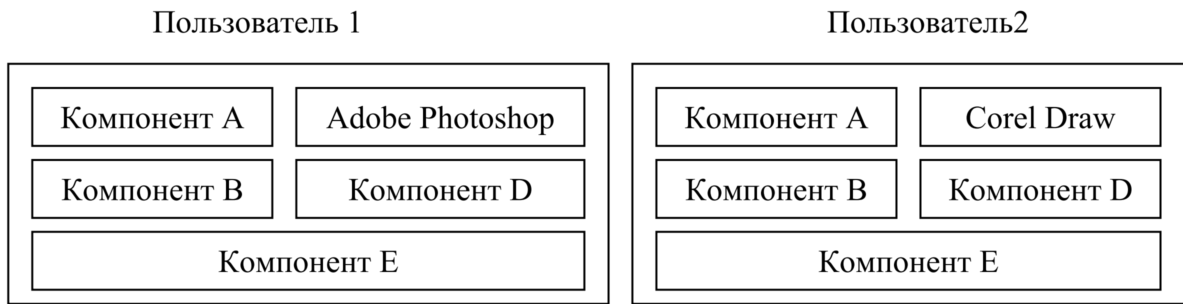


Рис. 2.3 *Создание приложений из компонентов упрощает адаптацию. Пользователь 1 предпочитает* Adobe Photoshop*, а пользователь 2 —* Corel Draw.

Библиотеки компонентов

Одна из самых многообещающих сторон внедрения компонентной архитектуры — быстрая разработка приложений. Если наши ожидания сбудутся, Вы сможете выбирать компоненты из библиотеки и составлять из них, как из деталей конструктора, цельные приложения (рис. 2.4).

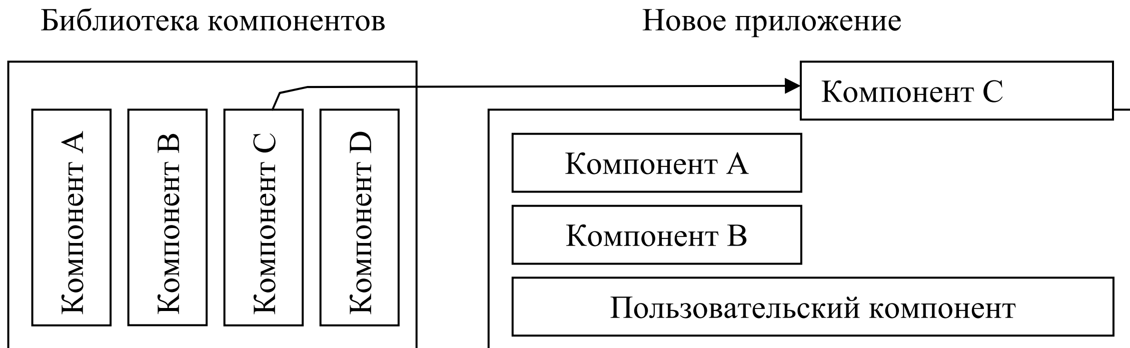


Рис. 2.4 *Из компонентов создаются библиотеки, используя которые, можно быстро разрабатывать приложения*

Сборка приложений из стандартных блоков давно была заветной мечтой программистов. Этот процесс уже начался с созданием управляющих элементов ActiveX (ранее известных как управляющие элементы OLE). Программисты на Visual Basic, C, C++ и Java могут воспользоваться управляющими элементами ActiveX для ускорения разработки своих приложений и страниц Web. Конечно, каждому приложению по-прежнему будут нужны и некоторые специализированные компоненты, но в основном можно будет обойтись стандартными.

Распределенные компоненты

С возрастанием производительности и общего значения сетей потребность в приложениях, состоящих из разбросанных по разным местам кусков, будет только повышаться. Компонентная архитектура помогает упростить процесс разработки подобных распределенных приложений. Приложения клиент-сервер — это уже шаг в сторону компонентной архитектуры, поскольку они разделены на две части, клиентскую и серверную.

Создать из обычного приложения распределенное легче, если это обычное приложение состоит из компонентов. Во-первых, оно уже разделено на функциональные части, которые могут располагаться вдали друг от друга. Во-вторых, поскольку компоненты заменяемы, вместо некоторого компонента можно подставить другой, единственной задачей которого будет обеспечивать связь с удаленным компонентом. Например, на рис.2.5 компонент C и компонент D расположены в сети на двух удаленных машинах. На локальной машине они заменяются двумя новыми компонентами, переадресовщиками C и D. Последние переправляют запросы от других компонентов к удаленным C и D по сети. Для приложения на локальной машине неважно, что настоящие компоненты C и D где-то в другом месте. Точно так же для самих удаленных компонентов не имеет значения их
расположение. При наличии подходящих переадресующих компонентов приложение может совершенно игнорировать фактическое местоположение своих частей.

Компонентное приложение с удаленными компонентами

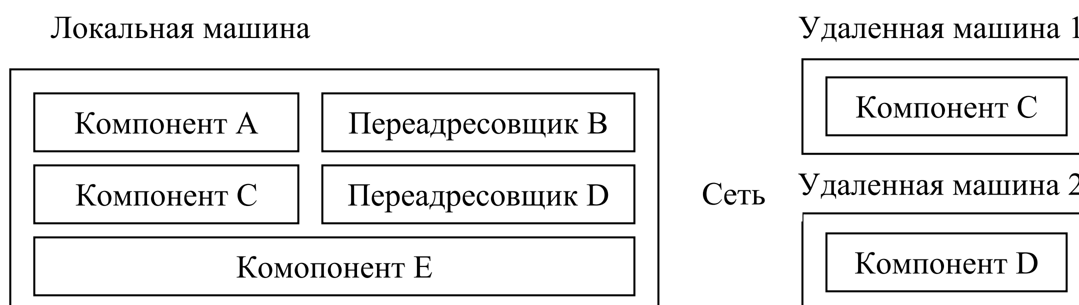


Рис. 2.5 *Расположение компонентов на удаленных машинах в сети*

Теперь, когда мы познакомились с некоторыми достоинствами компонентов, посмотрим, что требуется для их создания. Затем мы остановимся на роли, которую в создании компонентов играет COM.

Требования к компонентам

Преимущества использования компонентов непосредственно вытекают из способности последних подключаться к приложению и отключаться от него. Для этого компоненты должны удовлетворять двум требованиям. Во-первых, они должны компоноваться динамически. Во-вторых, они должны скрывать (или *инкапсулировать*) детали своей реализации. Попытка определить, какое из этих требований важнее, приводит к дилемме курицы и яйца. Каждое из этих требований зависит от другого. Я склонен считать, что критически важна динамическая компоновка, а сокрытие информации есть ее необходимое условие. Давайте рассмотрим эти требования более подробно.

Динамическая компоновка

Наша конечная цель — предоставить пользователю возможность заменять компоненты во время работы приложения. Хотя эта возможность не всегда реализуется, хотелось бы иметь для нее поддержку. Поддержка замены компонента во время выполнения требует динамической компоновки.

Чтобы понять, как это важно, лучше всего представить себе приложение, построенное из компонентов, которые не могут объединяться во время выполнения. Если

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

Инкапсуляция

Теперь давайте разберемся, почему динамическая компоновка требует инкапсуляции. Чтобы сформировать приложение, компоненты подключаются друг к другу. Если Вы хотите заменить один из компонентов, надо отключить от системы старый и подсоединить новый. Новый компонент должен подсоединяться тем же способом, что и старый, иначе компоненты придется переписывать, перекомпилировать и перекомпоновывать. Неважно, поддерживают ли компоненты и приложение динамическую компоновку, Вы разрушаете всю систему и должны перекомпилировать, если не переписывать, все заново.

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

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

Необходимость изоляции клиента от деталей реализации накладывает на компоненты ряд важных ограничений.
Ниже приведен список этих ограничений:

1. Компонент должен скрывать используемый язык программирования. Любой клиент должен иметь возможность использовать компонент, независимо от языков программирования, на которых написаны тот или другой. Раскрытие языка реализации создает новые зависимости между клиентом и компонентом.
2. Компоненты должны распространяться в двоичной форме. Действительно, поскольку они должны скрывать язык реализации, их необходимо поставлять уже скомпилированными, скомпонованными и готовыми к использованию.
3. Должна быть возможность модернизировать компоненты, не затрагивая уже существующих пользователей. Новые версии компонента должны разработать как с новыми, так и со старыми клиентами.
4. Должна быть возможность прозрачно перемещать компоненты в сети. Необходимо, чтобы компонент и использующая его программа могли

выполняться внутри одного процесса, в разных процессах или на разных машинах. Клиент должен рассматривать удаленный компонент так же как и локальный. Если бы с удаленными компонентами надо было работать иначе, чем с локальными, то потребовалось бы перекомпилировать клиент всякий раз, когда локальный компонент перемещается в другоеместо сети.

Независимость от языка

Многие не считают нужным требовать от компонентов независимости от языка программирования, как это сделано выше. Для обоснования своей позиции предположим, что у нас есть приложение, которое можно настраивать и перестраивать только при помощи компонентов, написанных на Objective C. Желающих писать для нас компоненты найдется немного, так как большинство программистов пользуются C++. Через некоторое время мы поймем, что никто не собирается писать для нашего приложения компоненты, и зададим в качестве рабочего языка C++. В результате у приложения появится значительно больше компонентов. Однако тут родится мода на новый язык, скажем, EspressoBeans, и все перейдут на него, оставляя компиляторы C++ пылиться без дела. Чтобы не сойти с дистанции, мы тоже потребуем писать компоненты на EspressoBeans. Итак, в этому моменту у нас уже будет три совершенно разных варианта создания компонентов для нашего приложения. Тут мы как раз и разоримся. Оказывается, в нашем сегменте рынка доминирует Visual Basic. Наш конкурент предоставил своим клиентам возможность писать компоненты на любом языке, включая Visual Basic, и по-прежнему процветает.
Если архитектура не зависит от языка, то компоненты может писать кто угодно. Они не будут устаревать при появлении новых языков. Такая архитектура обеспечит нам успех на рынке.

Версии

Пользователь может иметь два клиентских приложения, использующих один и тот же компонент. Предположим, что одно приложение применяет новую версию этого компонента, а другое — старую. Установка новой версии не должна нарушить работу приложения, которое использовало старую версию. На рис. 2.6 старое приложение использует новый компонент *Photoshop* абсолютно так же, как это делает новое.
Однако обратная совместимость не должна ограничивать развитие компонентов. Нужно, чтобы поведение компонента для новых приложений можно было радикально изменять, не нарушая поддержку старых приложений.

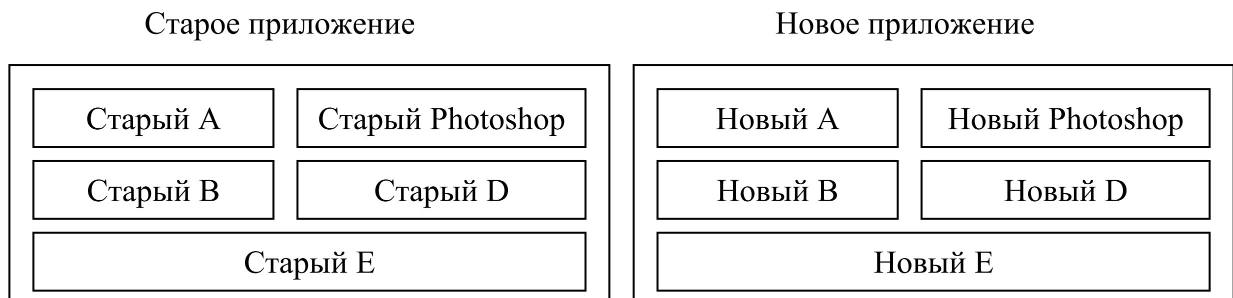


Рис. 1.6 *От новых компонентов требуется не нарушать работу старых и улучшать работу новых версий других компонентов*

COM

COM — это спецификация. Она указывает, как создавать динамически взаимозаменяемые компоненты. COM определяет стандарт, которому должны следовать компоненты и клиенты, чтобы гарантировать возможность совместной работы. Стандарты важны для компонентных архитектур так же, как и для любой системы с взаимозаменяемыми частями. Если бы не было стандарта на видеокассеты VHS, то найти подходящую ленту к Вашему видеомагнитофону было бы редкой удачей. Стандарты определяют диаметры резьбы для садовых шлангов и водопроводных кранов, на которые шланги надевают. Стандартам следуют платы PCMCIA и разъемы под них. Сигнал, принимаемый телевизором или радиоприемником, подчиняется стандарту. Стандарты особенно важны, когда разные части системы разрабатывают разные люди из разных организаций в разных странах. Без стандартов ничто не могло бы работать вместе. И у Microsoft есть внутренние стандарты, которым мы следуем при программировании (по крайней мере, большей частью).

Спецификация COM (COM Specification) — это документ, который устанавливает стандарт для нашей компонентной архитектуры. Компоненты, которые мы будем разрабатывать в этой книге, следуют данному стандарту. Спецификация COM содержится на сайте Microsoft. Однако Вы, вероятно, уже хотите точно знать, что же такое компоненты COM.

Компоненты COM — это…

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

Компоненты COM подключаются друг к другу динамически. Для этой цели COM использует DLL. Но, как мы видели, сама по себе динамическая компоновка не обеспечивает компонентной архитектуры. Компоненты должны быть инкапсулированы.

Инкапсуляция в компонентах COM достигается легко, поскольку они удовлетворяют нашим ограничениям:

- Компоненты COM полностью независимыот языка программирования. Они могут быть разработаны с помощью практически любого процедурного языка, включая Ada, C, Java, Modula-3, Oberon и Pascal.
- Любой язык, в том числе Smalltalk и Visual Basic, можно приспособить к использованию компонентов COM. Можно даже написать компоненты COM, используемые из языков макрокоманд.
- Компоненты COM могут распространяться в двоичной форме.
- Компоненты COM можно модернизировать, не нарушая работы старых клиентов.
- Компоненты COM можно прозрачно перемещать по сети. Компонент на удаленной системе рассматривается так же, как компонент на локальном компьютере.

Компоненты COM объявляют о своем присутствии стандартным способом. Используя схему объявлений COM, клиенты могут динамически находить нужные им компоненты.

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

Про многие вещи можно сказать «это — COM», но есть и много вещей, которые к COM не относятся, хотя их часто путают.

COM — это не…

COM — это не язык программирования. COM — не конкурент языкам программирования. Спор о том, что лучше — C++ или COM не имеет смысла; у них разное назначение. COM указывает нам, как писать компоненты. При этом мы свободны в выборе языка.

COM не является также конкурентом или заменой DLL. COM использует их для динамического объединения компонентов. Однако, по моему мнению, COM дает наилучший способ использовать возможности DLL. Любая проблема, которую можно решить при помощи DLL, лучше решается с компонентами COM. Я бы вообще не использовал DLL иначе как для поддержки компонентов COM. Это показывает, как эффективно COM использует DLL.

COM, по своей сути, — это не API или набор функций, подобный API Win32. COM не предоставляет таких сервисов, как *MoveWindow* и т.п. (Тем не менее, COM предоставляет некоторые сервисыуправления компонентами, которые описаны ниже.) Напротив, COM — это способ создания компонентов, которые могут предоставлять сервисы через объектно-ориентированные API. COM — и не библиотека классов C++, подобная MFC. COM обеспечивает способ разработки библиотек компонентов, независимых от языка программирования, но не занимается собственно разработкой.

Библиотека COM

COM — это больше, чем просто спецификация: в состав COM входит и некоторое количество кода. В COM есть API; это библиотека COM, предоставляющая сервисы управления компонентами, которые полезны всем клиентам и компонентам. Большинство функций этого API несложно реализовать самостоятельно, если Вы разрабатываете компоненты в стиле COM не для Windows. Библиотека COM создана, чтобы гарантировать единообразное выполнение всеми компонентами наиболее важных операций. Она экономит время разработчикам, создающим собственные компоненты и клиенты. Большая часть кода в библиотеке COM служит для поддержки распределенных или сетевых компонентов. Реализация распределенной COM (Distributed COM, DCOM) в системах Windows предоставляет код, необходимый для обмена информацией с компонентами по сети. Это избавляет Вас не только от необходимости писать такой код, но и от необходимости знать, как это делать.

Стиль COM

Самое лучшее определение для COM – это стиль программирования. Этот стиль можно использовать в рамках любой операционной системы и, любого языка программирования. Для этого Вам не нужен никакой Windows-специфичный код COM. Примеры в первых восьми главах этой книги легко модифицировать так, чтобы вообще не использовать код Windows. COM — это воплощение концепций компонентного программирования. Подобно структурному или объектно-ориентированному программированию, COM — это способ организации программ. Концепции правильного проектирования программ входят в спецификацию COM как составная часть.

COM дает больше, чем необходимо

COM удовлетворяет всем требованиям к компонентной архитектуре, которые обсуждались ранее. COM использует DLL для поддержки компонентов, которые могут взаимозаменятьсяво время выполнения. COM гарантирует, что такие компоненты могут воспользоваться всеми преимуществами динамической компоновки, посредством:

- установления стандарта, которому должны следовать компоненты;
- практически прозрачной поддержки нескольких версий компонента;
- обеспечения возможности работы со сходными компонентами одинаковым способом;
- определения архитектуры, независимой от языка;
- поддержки прозрачных связей с удаленными компонентами.

COM обеспечивает строгое отделение клиента от компонента. Именно в этой строгой изоляции заключена сила COM.

3. Интерфейс

Интерфейс обеспечивает соединение двух разных объектов. Для стыковки в компьютерных программах применяются наборы функций. Именно такие наборы и определяют интерфейс между разными частями программы.

Интерфейс DLL — это набор функций, экспортируемых ею. Интерфейс класса C++ — это набор членов данного класса. Интерфейс СОМ также включает в себя набор функций, которые реализуются компонентами и используются клиентами. Но СОМ дает более точное определение интерфейса. В СОМ интерфейсом является определенная структура в памяти, содержащая массив указателей на функции. Каждый элемент массива содержит адрес функции, реализуемой компонентом. С точки зрения СОМ интерфейсом является упомянутая структура в памяти; все остальное — это детали реализации, которые СОМ не касаются. Мы остановимся на этой структуре в конце главы, после того как посмотрим на ее реализацию с помощью C++.

На C++ интерфейс реализуется с помощью абстрактных классов. Поскольку каждый компонент СОМ может поддерживать сколь угодно много интерфейсов, для реализации компонента с несколькими интерфейсами мы будем использовать множественное наследование абстрактных базовых классов.

Интерфейсы — это все

В СОМ интерфейсы — это все. Для клиента компонент представляет собой набор интерфейсов. Клиент может взаимодействовать с компонентом СОМ только через интерфейс. Клиент очень мало знает о компоненте в целом. Часто ему даже не известны все интерфейсы, которые тот поддерживает.

На рисунках в гл. 2 были представлены приложения, построенные из компонентов (эти рисунки напоминают левую часть рис. 3.1). Они могут ввести Вас в заблуждение, так как подчеркивают важность компонентов и не обращают внимания на интерфейсы. С точки зрения программиста СОМ, интерфейсы — важная часть любого приложения. Компоненты сами по себе есть просто детали реализации интерфейсов. Правая часть рис. 3.1 более точно отражает значение интерфейсов для программ СОМ.

Повторное использование архитектур приложений

Утверждение, что компонент — всего лишь деталь реализации интерфейса, конечно, преувеличение. В конце концов, интерфейс без реализации ничего не сделает. Однако компонент можно удалить и заменить другим; если новый компонент поддерживает те же интерфейсы, что и старый, приложение будет работать по-прежнему.

Отдельные компоненты сами по себе не определяют приложения. Приложение определяют интерфейсы между компонентами. Пока интерфейсы неизменны, компоненты могут появляться и исчезать.

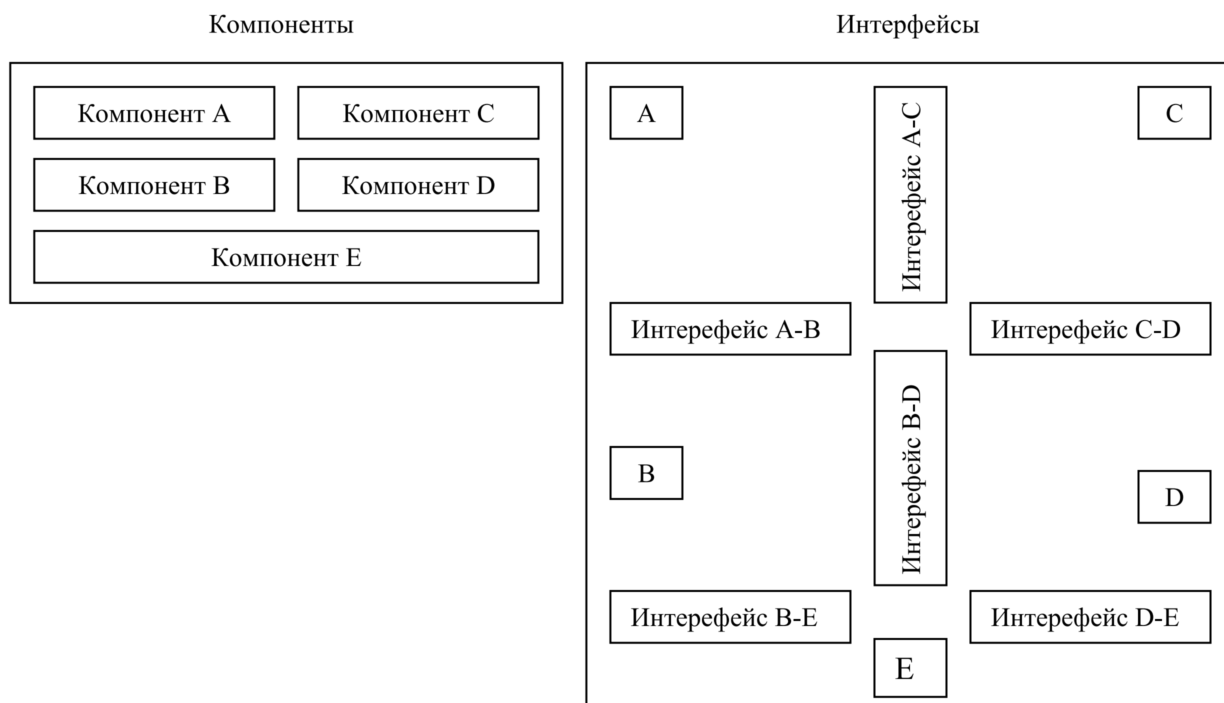


Рис. 3.1 *В COM интерфейсы значат больше, чем реализующие их компоненты*

Интерфейсы очень похожи на элементы каркаса сборного дома. Каркас задает структуру, без которой крыша и стены не защитят от стихии. Если Вы не трогаете каркас, дом остается структурно тем же самым. Если заменить кирпичные стены на деревянные, изменится внешний вид, но не структура. Аналогично этому, замена компонентов может изменить поведение приложения, но не его архитектуру. Одно из самых больших преимуществ компонентной модели — возможность повторного использования архитектуры приложения. При помощи тщательно разработанных интерфейсов можно создать архитектуры с очень высокой степенью пригодности к повторному использованию. Просто разрешив заменять некоторые ключевые компоненты, мы добиваемся того, что одна и та же архитектура может поддерживать несколько различных приложений. Однако построение повторно используемой архитектуры — непростая задача. В некотором смысле для этого надо уметь предсказывать будущее, что вряд ли многим по силам. Тем не менее, есть и другие причины для использования интерфейсов.

Другие преимущества интерфейсов COM

Интерфейсы позволяют клиенту работать с разными компонентами единообразно. Эта способность к унифицированной работе с разными компонентами известна как *полиморфизм (polymorphism).* Интерфейс обеспечивает описанные выше преимущества путем инкапсуляции определенного поведения.

Реализация интерфейса COM

Теперь рассмотрим код, реализующий простой интерфейс. В приведенном ниже тексте программы компонент *CA* использует *IX* и *IY* для реализации двух интерфейсов.

```
class IX // Первый интерфейс
{
public:
  virtual void Fx1() = 0;
  virtual void Fx2() = 0;
};

class IY // Второй интерфейс
{
public:
  virtual void Fy1() = 0;
  virtual void Fy2() = 0;
};

class CA : public IX, public IY // Компонент
{
public:
  // Реализация абстрактного базового класса IX
  virtual void Fx1() { cout << "Fx1" << endl; }
  virtual void Fx2() { cout << "Fx2" << endl; }
  // Реализация абстрактного базового класса IY
  virtual void Fy1() { cout << "Fy1" << endl; }
  virtual void Fy2() { cout << "Fy2" << endl; }
};
```

IX и *IY* — это *чисто абстрактные базовые классы*, которые используются для реализации интерфейсов. *Чисто абстрактный базовый класс (pure abstract base class)* — это базовый класс, который содержит только *чисто виртуальные функции (pure virtual functions)*. Чисто виртуальная функция — это виртуальная функция, «помеченная *=0* — знаком *спецификатора чистоты (pure specifier)*. Чисто виртуальные функции не реализуются в классах, в которых объявлены. Как видно из приведенного выше примера, функции *IX::Fx1, IX::Fx2, IY::Fy1* и *IY::Fy2* только декларируются. Реализуются же они в производном классе. В приведенном фрагменте кода компонент *CA* наследует два чисто абстрактных базовых класса — *IX* и *IY* — и реализует их чисто виртуальные функции.

Для того, чтобы реализовать функции-члены *IX* и *IY*, *CA* использует множественное наследование. Последнее означает, что класс является производным более чем от одного базового класса. Класс С++ чаще всего использует единичное наследование, т.е. имеет только один базовый класс. Далее в этой главе мы более подробно поговорим о множественных интерфейсах и множественном наследовании.
Абстрактный базовый класс напоминает канцелярский бланк, а производный класс заполняет этот бланк. Абстрактный базовый класс определяет функции, которые будет предоставлять производный класс, а производные классы реализуют их. Открытое (public) наследование от чисто абстрактного базового класса называется *наследованием интерфейса (interface inheritance)*, так как производный класс наследует лишь описания функций. Абстрактный базовый класс не содержит никакой реализации, которую можно было бы унаследовать.

В этом пособии все интерфейсы будут реализованы при помощи чисто абстрактных базовых классов. Поскольку COM не зависит от языка программирования, имеется двоичный стандарт того, что в этой модели считается интерфейсом. В последнем разделе данной главы — «Что за интерфейсом» — представлена эта структура. По счастью, многие компиляторы С++ автоматически генерируют в памяти верную структуру, если использовать чисто абстрактные базовые классы.

IX и *IY* не совсем интерфейсы в смысле COM. Чтобы стать настоящими интерфейсами, *IX* и *IY* должны наследовать специальный интерфейс *IUnknown*. Однако *IUnknown* — это предмет следующей главы. До конца данной главы мы будем считать, что *IX* и *IY* — это интерфейсы COM.

Законченный пример

Давайте рассмотрим несложную, но законченную реализацию интерфейсов *IX* и *IY*. Для реализации компонентов мы используем простую программу на C++ без динамической компоновки. Динамическую компоновку мы добавим в гл. 5, а пока гораздо проще обойтись без нее. В листинге 2-1 класс *CA* реализует компонент, который поддерживает интерфейсы *IX* и *IY*. В качестве клиента в этом примере выступает процедура *main*.

IFACE.CPP

```
//
// Iface.cpp
//
#include <iostream.h>
#include <objbase.h> // Определить интерфейс

void trace(const char* pMsg) { cout << pMsg << endl; }

// Абстрактные интерфейсы
interface IX
{
  virtual void __stdcall Fx1() = 0;
  virtual void __stdcall Fx2() = 0;
};
interface IY
{
  virtual void __stdcall Fy1() = 0;
  virtual void __stdcall Fy2() = 0;
};

// Реализация интерфейса
class CA : public IX,
public IY
{
  public:
  // Реализация интерфейса IX
  virtual void __stdcall Fx1() { cout << "CA::Fx1" << endl; }
  virtual void __stdcall Fx2() { cout << "CA::Fx2" << endl; }
  // Реализация интерфейса IY
  virtual void __stdcall Fy1() { cout << "CA::Fy1" << endl; }
  virtual void __stdcall Fy2() { cout << "CA::Fy2" << endl; }
};

// Клиент
int main()
{
  trace("Клиент: Создание экземпляра компонента");
  CA* pA = new CA;
  IX* pIX = pA; // Получить указатель IX
  trace("Клиент: Использование интерфейса IX");
  pIX->Fx1();
  pIX->Fx2();
  IY* pIY = pA; // Получить указатель IY
  trace("Клиент: Использование интерфейса IY");
  pIY->Fy1();
  pIY->Fy2();
  trace("Клиент: Удаление компонента");
  delete pA;
return 0;
}
```

Листинг 3.1 *Полный пример использования интерфейсов*

Результаты работы этой программы таковы:

Клиент: Создание экземпляра компонента
Клиент: Использование интерфейса IX
```
CA::Fx1
CA::Fx2
```
Клиент: Использование интерфейса IY
```
CA::Fy1
CA::Fy2
```
Клиент: Удаление компонента

Как видно из текста, клиент и компонент взаимодействуют через два интерфейса. Последние реализованы с помощью двух чисто абстрактных базовых классов *IX* и *IY*. Компонент реализуется классом *CA*, который наследует как *IX*, так и *IY*. Класс *CA* реализует функции-члены обоих интерфейсов.

Клиент создает экземпляр компонента. Далее он получает указатели на интерфейсы, поддерживаемые компонентом. Потом клиент использует эти указатели совершенно аналогично указателям на классы C++, поскольку интерфейсы реализованы как чисто абстрактные базовые классы. Ключевыми в этом примере являются следующие моменты:

- Интерфейсы COM реализуются как чисто абстрактные базовые классы C++
- Один компонент COM может поддерживать несколько интерфейсов
- Класс C++ может использовать множественное наследование для реализации компонента, поддерживающего несколько интерфейсов.

Взаимодействие в обход интерфейсов

Ранее упоминалось что клиент и компонент взаимодействуют только через интерфейс. Клиент из листинга 3.1 не следует этому правилу. Он взаимодействует с компонентом посредством *pA* — указателя на класс *CA*, а не на интерфейс. Это может показаться несущественным, но на самом деле очень важно. Использование указателя на *CA* требует, чтобы клиент знал, как объявлен (обычно в заголовочном файле) класс *CA*. Объявление класса содержит множестве деталей реализации. Изменение этих деталей потребует перекомпиляции клиента. Компоненты должны уметь добавлять и удалять интерфейсы без нарушения работы старых клиентов. Это одна из причин, по которым мы настаиваем, что клиент и компонент должны взаимодействовать только через интерфейсы. Вспомните, что интерфейсы основаны на чисто абстрактных базовых классах, с которыми не связана какая-либо реализация.

Конечно, не обязательно изолировать клиент от компонента, если они находятся в одном файле. Однако подобная изоляция необходима, если клиент и компонент подключаются друг к другу динамически, особенно когда у Вас нет исходных текстов. В гл. 4 мы исправим наш пример так, чтобы в нем не использовался указатель на *CA*. Клиенту более не потребуется знать как объявлен класс *CA*.

Использование указателя на *CA* — не единственное место, где клиент из предыдущего примера в обход интерфейса взаимодействует с компонентом. Для управления существованием компонента клиент применяет операторы *new* и *delete*. Эти операторы не только не входят ни в один из интерфейсов, но и специфичны для языка C++. В гл. 5 мы рассмотрим, как удалить компонент через интерфейс без помощи специфичного для языка оператора. В гл. 7 и 8 будет рассмотрен гораздо более мощный способ создания компонентов.

Класс — это не компонент

В 3.1 класс *CA* реализует один компонент. COM не требует, чтобы один класс C++ соответствовал одному компоненту. Вы можете реализовать один компонент при помощи нескольких классов. На самом деле компонент можно реализовать вообще без классов. Классы C++ не используются при реализации компонентов COM на С, а потому они не обязательно должны использоваться и на C++. Просто компоненты COM гораздо легче реализовать через классы, чем строить вручную.

Интерфейсы не всегда наследуются

Класс *CA* наследует поддерживаемые им интерфейсы. COM не требует, чтобы класс, реализующий интерфейсы, наследовал их, поскольку клиент никогда не видит иерархию наследования в компоненте COM. Наследование интерфейсов — это в чистом виде деталь реализации. Вместо того, чтобы собирать все интерфейсы в одном классе, Вы можете реализовать их отдельно и использовать указатели на соответствующие классы. Крейг Брокшмидт использует такой метод в своей книге *Inside OLE*. Мы будем использовать для реализации всех интерфейсов один класс, поскольку этот метод проще и легче для восприятия; он также делает программирование для COM на C++ более естественным.

Множественные интерфейсы и множественное наследование

Компоненты могут поддерживать сколь угодно много интерфейсов. Для поддержки нескольких интерфейсов мыиспользуем множественное наследование. В листинге 3.1 *CA* является производным от двух интерфейсов *IX* и *IY*, которые он поддерживает. Благодаря поддержке множественных интерфейсов компонент можно рассматривать как набор интерфейсов.

Это определяет рекурсивно-вложенную природу компонентной архитектуры (см. рис. 3.2). Интерфейс — это набор функций, компонент — набор интерфейсов, а система — набор компонентов. Некоторые считают интерфейсы эквивалентами функциональных возможностей и при добавлении к компоненту новых интерфейсов говорят о появлении новых возможностей. Я же предпочитаю рассматривать интерфейсы как различные варианты поведения компонента. Набор интерфейсов соответствует набору таких вариантов.

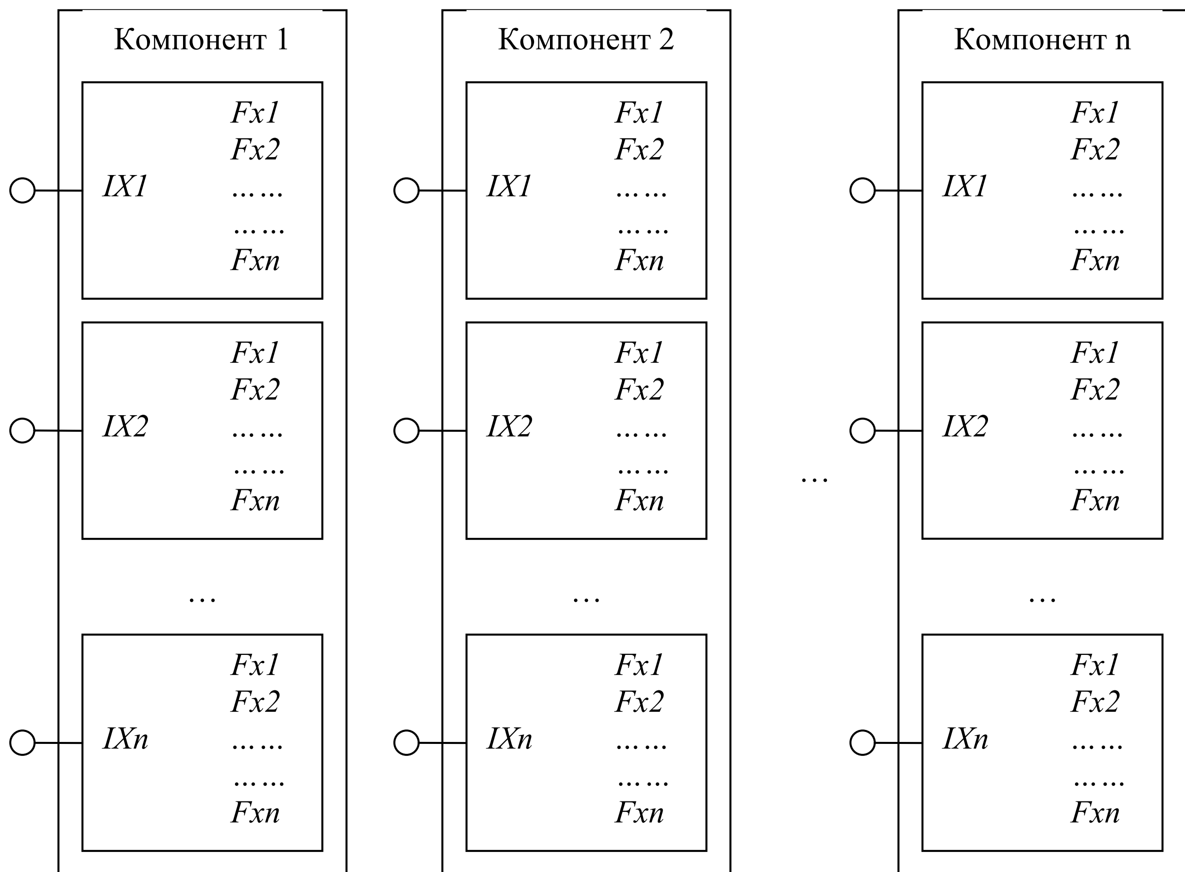


Рис. 3.2 *Система компонентов — это набор компонентов, из которых каждый поддерживает набор интерфейсов, из которых каждый содержит набор функций*

Конфликт имен

Поскольку компонент может поддерживать несколько интерфейсов, легко возникает конфликт имен функций этих интерфейсов. Действительно, СОМ нет до этого дела. Интерфейсы СОМ отвечают двоичному стандарту; клиент подключается к интерфейсу не по имени самого интерфейса или его функций. Клиент обращается к функции интерфейса по ее адресу в блоке памяти, представляющем интерфейс. Структура этого блока памяти будет рассмотрена в конце данной главы. Техника изменения имен интерфейсов и функций обсуждается в гл. 9.

Другое решение проблемы конфликта имен — отказ от множественного наследования. Классу, управляющему компонентом, нет необходимости наследовать каждый интерфейс; он может содержать указатели на другие классы, которые реализуют отдельные интерфейсы.

Интерфейсы не изменяются

Хотя из листинга 3.1 это не очевидно, интерфейсы нельзя изменять, причем нельзя изменять никогда. Это один из наиболее шокирующих моментов СОМ. После того, как интерфейс опубликован, он обязан быть всегда тем же самым. При модификации компонента следует не изменять существующий интерфейс, а создать и добавить новый. Множественные интерфейсы позволяют компоненту поддерживать новые интерфейсы в дополнение к старым. Таким образом, множественные интерфейсы обеспечивают

прочный фундамент, на основе которого клиенты и компоненты могут воспринимать новые версии партнеров. Проблему версий мы затронем в следующей главе.

Полиморфизм

Полиморфизм — это способность обрабатывать разные объекты единообразно. Поддержка множественных интерфейсов обеспечивает дополнительные возможности для полиморфизма. Если два разных компонента поддерживают один и тот же интерфейс, для работы с ними клиент может использовать один и тот же код. Таким образом, клиент может работать с разными компонентами полиморфно.

Множественные интерфейсы поощряют полиморфизм. Чем больше интерфейсов поддерживает компонент, тем меньше может быть каждый из них. Маленький интерфейс представляет один вариант поведения, а большой — несколько вариантов. Чем больше вариантов поведения поддерживает данный интерфейс, тем более специфичным для конкретной ситуации он становится. Чем специфичнее интерфейс, тем меньше вероятность его повторного использования другим компонентом. Если же интерфейс не используется повторно, то и работающий с ним клиентский код также нельзя использовать повторно.

Например, что имеет больший потенциал повторного применения — единичный интерфейс, представляющий поведение вертолета и описывающий полет, зависание, подъем, вращение, вибрацию, удары и падение, или несколько интерфейсов, реализующих отдельные варианты поведения? Интерфейс, представляющий полет, имеет гораздо больше шансов быть повторно использованным, чем интерфейс «вертолета вообще». Вряд ли что-либо, кроме вертолета, будет вести себя в точности как он; однако есть много аппаратов, которые летают.

Замечательный результат полиморфизма — возможность повторного использования всего приложения. Предположим, что Вы пишете приложение Viewer для просмотра растровых изображений (bitmap). Последние, реализуются как компоненты COM, поддерживающие интерфейс *IDisplay*. Viewer взаимодействует с компонентами только через этот интерфейс. Приходит Ваш начальник и говорит, что ему нужно просматривать файлы VRML. Вместо того, чтобы писать новую версию Viewer, Вы должны просто написать компонент COM, реализующий интерфейс *IDisplay*, но отображающий не растровые изображения, а файлы VRML. Конечно, написание программы отображения VRML потребует значительной работы, но не придется переписывать приложение целиком.

Возможность повторного применения целых архитектур не возникает автоматически. Она требует тщательного планирования при разработке интерфейсов, чтобы последние могли поддерживать много разных реализаций. Не только интерфейсы должны быть универсальными; и клиент должен использовать интерфейс универсальным образом, который не ограничивает возможности реализации интерфейса. Интерфейсы или приложения, не готовые к появлению новых компонентов, не могут воспользоваться всеми преимуществами полиморфизма и повторно применять целые конструкции. Кстати, в нашем примере маловероятно, что интерфейс *Idisplay* окажется достаточно универсальным и гибким для отображения файлов VRML в дополнение к растровым изображениям, если мы не запланируем это сразу. Одна из самых больших трудностей компонентного программирования — разработка повторно используемых, адаптируемых, гибких и готовых к будущим нововведениям интерфейсов.

Теперь, когда мы знаем, как реализовывать интерфейсы, и примерно представляем, на что они способны, давайте посмотрим, что же представляет собой интерфейс COM и почему для его реализации используют абстрактный базовый класс.

Что за интерфейсом

Сквозной сюжет этой главы сводится к тому, что интерфейсы COM реализуются на C++ при помощи чисто абстрактных базовых классов. В данном разделе объясняется, почему эти классы подходят для реализации интерфейсов. Мы увидим, что чисто абстрактный базовый класс определяет в памяти как раз структуру, которую COM требует от интерфейса.

Таблица виртуальных функций

Определяя чисто абстрактный базовый класс, мы фактически определяем структуру некоторого блока памяти. Все реализации чисто абстрактных базовых классов являются блоками памяти однотипной структуры. На рис. 3.3 показана структурапамяти для абстрактного базового класса, определяемого следующим кодом:

```
interface IX
{
  virtual void __stdcall Fx1() = 0;
  virtual void __stdcall Fx2() = 0;
  virtual void __stdcall Fx3() = 0;
  virtual void __stdcall Fx4() = 0;
};
```

Определение чисто абстрактного базового класса просто задает формат данных в памяти. Память для структуры не выделяется до тех пор, пока абстрактный класс не будет реализован в производном классе. Когда производный класс наследует абстрактный базовый класс, он наследует и эту структуру.

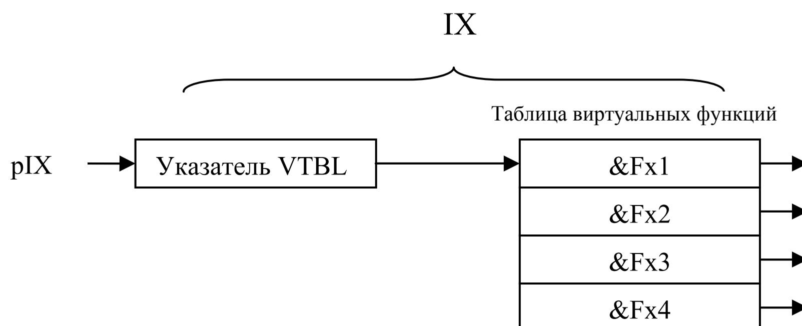


Рис. 3.3 *Пример структуры блока памяти, определяемой абстрактным базовым классом*

Блок памяти, определяемый чисто абстрактным базовым классом, состоит из двух частей. На рис. 3.3 справа показана *таблица виртуальных функций (virtual function table)*. Таблица виртуальных функций, или *vtbl* — это массив указателей на реализации виртуальных функций. На рисунке первый элемент vtbl содержит адрес функции *Fx1*, реализованной в производном классе. Второй элемент содержит адрес *Fx2*, и т.д. Слева на рисунке показан указатель на vtbl, ил просто указатель vtbl. Указатель на абстрактный базовый класс указывает на указатель vtbl, который, естественно, указывает на таблицу vtbl.

Оказывается, что формат блока памяти для интерфейса COM совпадает с форматом блока памяти, который компилятор C++ генерирует для абстрактного базового класса. Это значит, что для определения интерфейсов COM можно использовать абстрактные базовый

классы. Так, интерфейс *IX* — это и интерфейс, и абстрактный базовый класс. Он является интерфейсом COM, поскольку формат его структуры в памяти следует спецификации COM. Он является и абстрактным базовым классом, поскольку именно так мы его определили.

Конечно, гладко все выглядит только на бумаге. Компилятор C++ не обязан генерировать для абстрактного базового класса структуру, представленную на рис. 3.3. Ни один стандарт не задает такой формат. Причина отсутствия стандарта очевидна — программы на C++ используют один и тот же компилятор для всех исходных файлов, так что от него требуется совместимость только с самим собой. Однако так сложилось, что большинство компиляторов генерируют именно структуру, показанную на рис. 2-4. Все компиляторы C++ для Windows генерируют формат vtbl, подходящий для COM1.
Имеется дополнительное требование, которому должен удовлетворять интерфейс, чтобы он был интерфейсом COM. Все интерфейсы COM должны наследовать интерфейс *IUnknown*, до которого мы доберемся в следующей главе. Это означает, что первые три элемента vtbl одни и те же для всех интерфейсов COM. Они содержат адреса реализации трех функций-членов *IUnknown*. Подробнее об этом рассказывается в гл. 4.

Указатели vtbl и данные экземпляра

Зачем же нужен указатель vtbl? Указатель vtbl еще на ступеньку повышает уровень абстракции в процессе получения указателя на функцию по указателю на базовый класс. Это дает нам дополнительную свободу реализации интерфейса.

Компилятор C++ может генерировать код, в котором класс, реализующий абстрактный базовый класс, хранит вместе с указателем vtbl информацию, специфичную для экземпляра. Например, класс *CA* в представленном ниже коде реализует абстрактный базовый класс *IX*, определенный в предыдущем разделе.

```
class CA : public IX
{
public:
  // Реализация интерфейса IX
  virtual void __stdcall Fx1() { cout << “CA::Fx1” << endl; }
  virtual void __stdcall Fx2() { cout << m_Fx2 << endl; }
  virtual void __stdcall Fx3() { cout << m_Fx3 << endl; }
  virtual void __stdcall Fx4() { cout << m_Fx4 << endl; }

  // Конструктор
  CA(double d) : m_Fx2(d*d), m_Fx3(d*d*d), m_Fx4(d*d*d*d)
  {}

  // Данные экземпляра
  double m_Fx2;
  double m_Fx3;
  double m_Fx4;
};
```

Таблица vtbl и данные класса *CA*, сгенерированные компилятором, показаны на рис. 2-5. Обратите внимание, что данные экземпляра потенциально доступны через указатель класса *CA*. Однако обычно клиент не знает, какие именно данные там хранятся, и потому не может обращаться к ним.

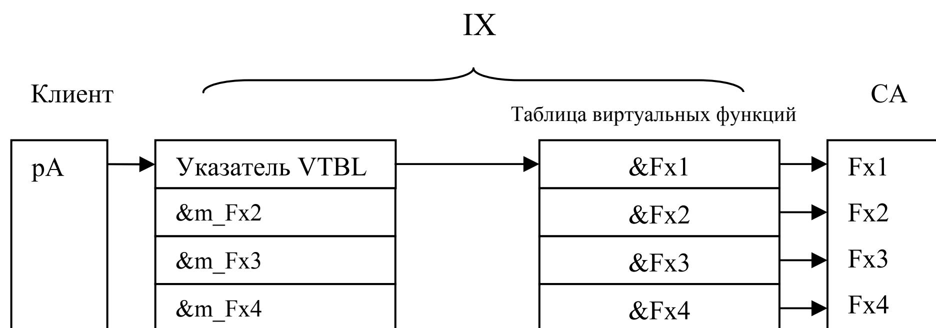


Рис. 3.4 *Данные, специфичные для экземпляра, хранятся вместе с указателем vtbl*

В то время как классы C++ могут обращаться к данным экземпляра напрямую, компоненты COM никогда не смогут добраться до них. В COM Вы можете работать с компонентом только через функции, и никогда — через переменные. Это соответствует тому способу, которым мы определяем интерфейсы COM. У чисто абстрактных базовых классовесть только чисто виртуальные функции, они не имеют данных экземпляра.

Множественные экземпляры

Однако указатель vtbl — это больше, чем просто удобное место для хранения данных экземпляра. Он также позволяет разным экземплярам одного класса использовать одну и ту же vtbl. Если мы создадим два экземпляра *CA*, то получим два отдельных набора данных экземпляра. Однако эти экземпляры могут совместно использовать одну и ту же vtbl и одну и ту же реализацию. Например, предположим, что мы создали два объекта *CA*:

```
int main()
{
  // Создать первый экземпляр CA
  CA* pA1 = new CA(1.5);
  // Создать второй экземпляр CA
  CA* pA2 = new CA(2.75);
…
}
```

Эти объекты могут использовать одну и ту же vtbl, элементы которой указывают на одни и те же реализации виртуальных функций-членов. Однако у объектов будут разные данные экземпляра (рис. 3.5).

Компоненты COM могут, но не обязаны применять указатель vtbl для совместного использования vtbl. Каждый экземпляр компонента COM может иметь свою vtbl.

Разные классы, одинаковые vtbl

По настоящему сила интерфейсов проявляется в том, что классы, производные от данного интерфейса, клиент может рассматривать одинаково. Предположим, что мы реализовали класс *CB*, который также является производным от *IX*:

```
class CB : public IX
{
public:
  // Реализация интерфейса IX
  virtual void __stdcall Fx1() { cout << "CB::Fx1" << endl; }
  virtual void __stdcall Fx2() { cout << "CB::Fx2" << endl; }
  virtual void __stdcall Fx3() { cout << "CB::Fx3" << endl; }
  virtual void __stdcall Fx4() { cout << "CB::Fx4" << endl; }
};
```

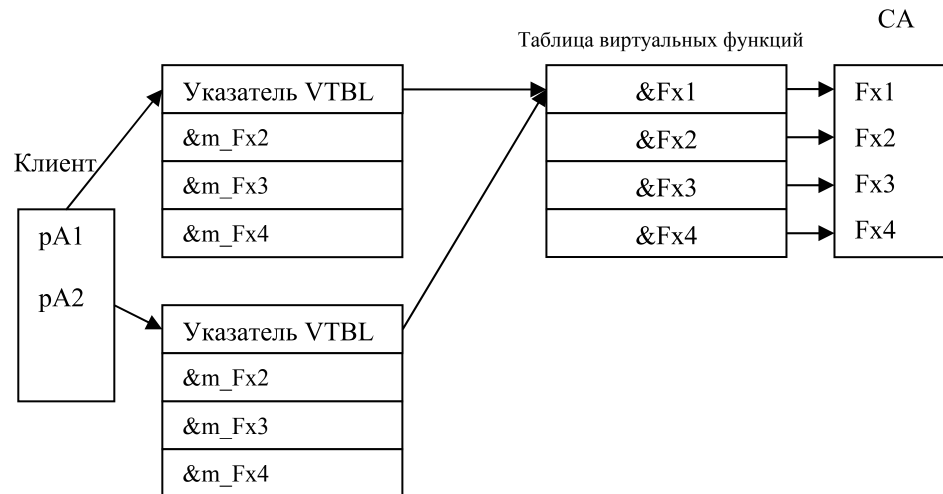


Рис. 3.5 *Несколько экземпляров класса используют одну vtbl*

С помощью указателя на *IX* клиент может работать как с *CA*, так и с *CB*:

```
void foo(IX* pIX)
{
  pIX->Fx1();
  pIX->Fx2();
}

int main ()
{
  // Создать экземпляр CA
  CA* pA = new CA(1.789);
  // Создать экземпляр CB
  CB* pB = new CB;
  // Получить указатель IX для CA
  IX* pIX = pA;
  foo(pIX);
  // Получить указатель IX для CB
  pIX = pB;
  foo(pIX);
…
}
```

В данном примере мы использовали и *CA*, и *CB* так, словно они являются интерфейсом *IX*. Это и есть полиморфизм. На рис. 3.6 показан формат структур памяти для данного примера. Не нарисованы данные экземпляров, поскольку нам как COM-программистам не важно, что они собой представляют.

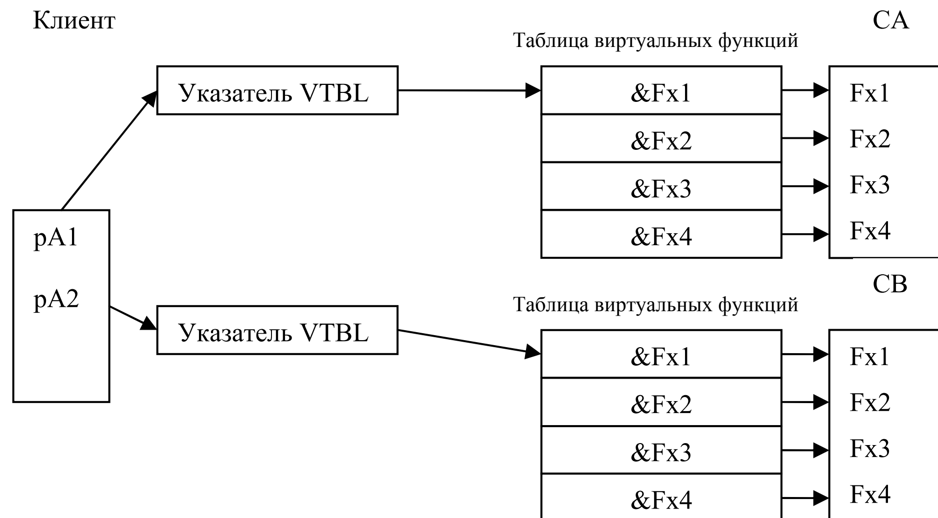


Рис. 3.6 *Полиморфное использование двух разных классов при помощи общего абстрактного базового класса*

Из рис. 3.6 видно, что два наших класса — *CA* и *CB* — имеют отдельные и различные данные экземпляра, vtbl и реализации. Однако доступ к их vtbl может осуществляться одинаково, поскольку формат обеих таблиц один и тот же. Адрес функции *Fx1* находится в первом элементе обеих таблиц, адрес *Fx2* — во втором, и т.д. Формат таблиц соответствует тому, который генерирует компилятор для абстрактного базового класса. Когда класс реализует абстрактный базовый класс, он обязуется следовать данному формату. То же самое верно для компонентов. Когда компонент возвращает указатель интерфейса *IX*, он обязан гарантировать, что тот указывает на корректную структуру.

Кирпичики COM, резюме

В этой главе мы превратили интерфейс из формальной концепции в конкретную структуру в памяти. Мы увидели, что благодаря инкапсуляции деталей реализации интерфейсы защищают систему компонентов от разрушительного влияния изменений. До тех пор, пока интерфейсы неизменны, клиент и компонент можно спокойно изменять. Благодаря этому старые компоненты можно заменять новыми, не нарушая работу системы. Клиенты также могут работать с компонентами, реализующими один и тот же интерфейс, полиморфно.

Кроме того, мы увидели, как реализовать интерфейс на C++ с помощью чисто абстрактного базового класса. Формат блока памяти, генерируемый компилятором C++ для чисто абстрактного базового класса, совпадает с определяемым COM форматом для интерфейса.

В этой главе Вы узнали, что такое интерфейс, как его реализовать и использовать. Однако приведенные в примерах интерфейсы — не настоящие интерфейсы COM. COM требует, чтобы все интерфейсы поддерживали три функции. Со ссылок на них начинается vtbl интерфейса. В следующей главе мы рассмотрим первую из этих трех функций — *QueryInterface.*

4. QueryInterface

Запрос интерфейса

Поскольку в СОМ все начинается и заканчивается интерфейсом, давайте и мы начнем с интерфейса, через который запрашиваются другие интерфейсы.

Клиент всегда взаимодействует с компонентом через некоторый интерфейс. Даже для запроса у компонента интерфейса используется специальный интерфейс *IUnknown*. Определение *IUnknown*, содержащееся в заголовочном файле UNKNWN.H, входящим в состав Win32 SDK, выглядит так:

```
interface IUnknown
{
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) = 0;
  virtual ULONG __stdcall AddRef() = 0;
  virtual ULONG __stdcall Release() = 0;
}
```

В *IUnknown* имеется функция с именем *QueryInterface*. Клиент вызывает ее, чтобы определить, поддерживает ли компонент некоторый интерфейс. В этой главе я собираюсь поговорить о *QueryInterface*. В гл. 5 мы рассмотрим *AddRef* и *Release*, которые предоставляют способ управления временем жизни интерфейса.

IUnknown

Мне всегда казалось забавным название *IUnknown*. Это единственный интерфейс, о котором знают все клиенты и компоненты, и тем не менее это «неизвестный интерфейс». Происхождение названия просто. Все интерфейсы СОМ должны наследовать *IUnknown*. Таким образом, если у клиента имеется указатель на *IUnknown*, то клиент, не зная, указателем на какой именно интерфейс обладает, знает, что может запросить через него другие интерфейсы.

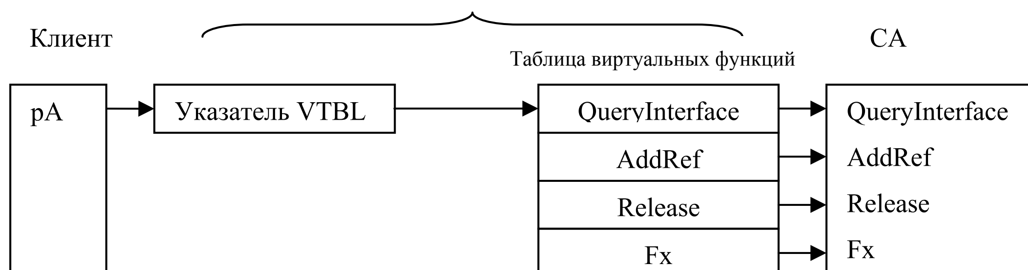


Рис. 4.1 *Все интерфейсы СОМ наследуют* IUnknown *и содержат указатели на* QueryInterface, AddRef *и* Release *в первых трех элементахсвоих vtbl*

Поскольку все интерфейсы СОМ наследуют *IUnknown*, в каждом интерфейсе есть функции *QueryInterface*, *AddRef* и *Release* — три первые функции в vtbl (см. рис. 4.1). Благодаря этому все интерфейсы СОМ можно полиморфно трактовать как интерфейсы *IUnknown*. Если в первых трех элементах vtbl интерфейса не содержатся указатели на три перечисленные функции, то это не интерфейс СОМ. Поскольку все интерфейсы наследуют *IUnknown*, постольку все они поддерживают *QueryInterface*. Таким образом, любой интерфейс можно использовать для получения всех остальных интерфейсов, поддерживаемых компонентом.

Поскольку все указатели интерфейсов являются также и указателями на *IUnknown*, клиенту не требуется хранить отдельный указатель на собственно компонент. Клиент работает только с указателями интерфейсов.

Получение указателя на *IUnknown*

Каким образом клиент может получить указатель на *IUnknown*? Мы используем функцию с именем *CreateInstance*, которая создает компонент и возвращает указатель на *IUnknown*:

```
IUnknown* CreateInstance();
```

Клиент использует *CreateInstance* вместо оператора *new*. В этой главе мы создадим простую версию данной функции, которую будем изменять на протяжении нескольких последующих глав в соответствии с нашими потребностями. В гл. 7 и 8 будет представлен «официальный» способ создания компонентов COM.

Теперь, когда мы знаем, как клиент получает указатель на *IUnknown*, давайте посмотрим, как он может использовать *QueryInterface* для получения других интерфейсов, а затем займемся реализацией *QueryInterface*.

Знакомство с *QueryInterface*

IUnknown содержит функцию-член *QueryInterface*, при помощи которой клиент определяет, поддерживается ли тот или иной интерфейс. *QueryInterface* возвращает указатель на интерфейс, если компонент его поддерживает; в противном случае возвращается код ошибки (тогда клиент может запросить указатель на другойинтерфейс или аккуратно выгрузить компонент).

У *QueryInterface* два параметра:

```
virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
```

Первый параметр — *идентификатор интерфейса*, так называемая IID-структура. Более подробно IID будут рассматриваться в гл. 7. Пока же мы будем рассматривать их как константы, задающие интерфейс. Второй параметр — адрес, по которому *QueryInterface* помещает указатель на искомый интерфейс.

QueryInterface возвращает HRESULT; это не описатель (handle), как может показаться по названию. HRESULT — просто 32-разрядный код результата, записанный в определенном формате. *QueryInterface* может возвратить либо *S_OK*, либо *E_NOINTERFACE*. Клиент не должен прямо сравнивать возвращаемое *QueryInterface* значение с этими константами; для проверки надо использовать макросы SUCCEEDED или FAILED. Исчерпывающее обсуждение HRESULT содержится в гл. 7.
Теперь посмотрим, как используется, а затем — как реализуется *QueryInterface*.

Использование *QueryInterface*

Предположим, что у нас есть указатель на *IUnknown*, *pI*. Чтобы определить, можно ли использовать некоторый другой интерфейс, мы вызываем *QueryInterface*, передавая ей идентификатор нужного нам интерфейса. Если *QueryInterface* отработала успешно, мы можем пользоваться указателем:

```
void foo(IUnknown* pI)
{
  // Определить указатель на интерфейс
  IX* pIX = NULL;

  // Запросить интерфейс IX
  HRESULT hr = pI->QueryInterface(IID_IX, (void**)&pIX);

  // Проверить значение результата
  if (SUCCEEDED(hr))
  {
    // Использовать интерфейс
    pIX->Fx();
  }
}
```

В этом фрагменте кода мы запрашиваем у *pI* интерфейс, идентифицируемый с помощью *IID_IX*. Определение *IID_IX* содержится в заголовочном файле, предоставляемом.

Обратите внимание, что *pIX* устанавливается в NULL перед вызовом *QueryInterface*. Это пример хорошего программирования с защитой от ошибок. Как мы вскоре увидим, предполагается, что для неудачного запроса *QueryInterface* должна устанавливать возвращаемый указатель в NULL. Однако, поскольку *QueryInterface* реализуется программистом компонента, в некоторых реализациях это наверняка не будет сделано. Для безопасности следует установить указатель в NULL самостоятельно.

Таковы основы использования *QueryInterface*. Позже мы рассмотрим некоторые более продвинутые приемы ее применения. Но сначала давайте посмотрим, как следует реализовывать *QueryInterface* в наших компонентах.

Реализация *QueryInterface*

Реализовать *QueryInterface* легко. Все, что нужно сделать, — это вернуть указатель интерфейса, соответствующего данному IID. Если интерфейс поддерживается, то функция возвращает S_OK и указатель. В противном случае возвращаются E_NOINTERFACE и NULL. Теперь давайте запишем *QueryInterface* для следующего компонента, реализуемого классом *CA*:

```
interface IX : IUnknown { /*...*/ };
interface IY : IUnknown { /*...*/ };
class CA : public IX, public IY { /*...*/ };
```

Иерархия наследования для этого класса и его интерфейсов показана на рис. 4.2.

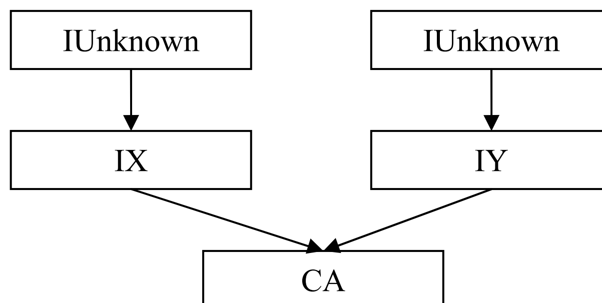


Рис. 4.2 *Иерархия наследования для приведенного выше фрагмента кода*

Невиртуальное наследование
Обратите внимание, что *IUnknown* — не виртуальный базовый класс. *IX* и *IY* не могут наследовать *Iunknown* виртуально, так как виртуальное наследование приводит к vtbl, несовместимой с форматом COM. Ели бы *IX* и *IY* наследовали *IUnknown* виртуально, то первые три элемента их vtbl не были бы указателями на три функции-члена *IUnknown*.

Следующий фрагмент кода реализует *QueryInterface* для класса из приведенного выше фрагмента кода. Эта версия функции возвращает указатели на три разных интерфейса — *IUnknown*, *IX* и *IY*. Обратите внимание, что возвращаемый указатель на *IUnknown* всегда один и тот же несмотря на то, что класс *CA* наследует два таких интерфейса (от *IX* и от *IY*).

```
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    // Клиент запрашивает интерфейс IX
    *ppv = static_cast<IX*>(this);
  }
  else if (iid = IID_IY)
  {
    // Клиент запрашивает интерфейс IY
    *ppv = static_cast<IY*>(this);
  }
  else
  {
    // Мы не поддерживаем запрашиваемый клиентом интерфейс.
    // Установить возвращаемый указатель в NULL.
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  static_cast<IUnknown*>(*ppv)->AddRef(); // См. гл. 4
  return S_OK;
}
```

Здесь для реализации *QueryInterface* использован простой оператор *if-then-else*. Вы можете использовать любой другой способ, обеспечивающий проверку и ветвление. Мне случалось встречать реализации на основе массивов, хэш-таблиц и деревьев; они полезны, когда компонент поддерживает много интерфейсов. Нельзя, однако, использовать оператор *case*, поскольку идентификатор интерфейса — структура, а не константа.

Обратите внимание, что *QueryInterface* устанавливает указатель интерфейса в NULL, если интерфейс не поддерживается. Это не только требование COM, это вообще полезно; NULL вызовет фатальную ошибку в клиентах, которые не проверяют возвращаемые значения. Это менее опасно, чем позволить клиенту выполнять произвольный код, содержащийся по неинициализированному указателю. Кстати, вызов *AddRef* в конце *QueryInterface* в настоящиймомент ничего не делает. Реализацией *AddRef* мы займемся в гл. 5.

Основы приведения типов

Вы, вероятно, заметили, что *QueryInterface* выполняет приведение указателя *this*, прежде чем сохранить его в *ppv*. Это очень важно. В зависимости от приведения,

значение, сохраняемое в *ppv*, может изменяться. Да-да, приведение *this* к указателю на *IX* дает не тот же адрес, что приведение к указателю на *IY*. Например:

```
static_cast<IX*>(this) != static_cast<IY*>(this)
static_cast<void*>(this) != static_cast<IY*>(this)
```

или, для тех, кому привычнее старый стиль,

```
(IX*)this != (IY*)this
(void*)this != (IY*)this
```

Изменение указателя *this* при приведении типа обусловлено тем, как в C++ реализовано множественное наследование. Более подробно об этом рассказывает врезка «Множественное наследование и приведение типов».

Перед присваиванием указателю, описанному как *void*, надо всегда явно приводить *this* к нужному типу. Интересная проблема связана с возвратом указателя на *IUnknown*. Можно было бы написать:

```
*ppv = static_cast<IUnknown*>(this); // неоднозначность
```

Однако такое приведение неоднозначно, поскольку *IUnknown* наследуют оба интерфейса, *IX* и *IY*. Таким образом, следует выбрать, какой из указателей — *static_cast<IUnknown*>(static_cast<IX*>(this))* или *static_cast<IUnknown*>(static_cast<IY*>(this))* — возвращать. В данном случае выбор не существенен, поскольку реализации указателей идентичны. Однако Вы должны действовать по всей программе единообразно, поскольку указатели не идентичны — а COM требует, чтобы для *IUnknown* всегда возвращался один и тот же указатель. Это требование будет обсуждаться далее в этой главе.

Множественное наследование и приведение типов

Обычно приведение указателя к другому типу не изменяет значения. Однако для поддержки множественного наследования C++ в некоторых случаях изменяет указатель на экземпляр класса. Большинство программистов на C++ не знают об этом побочном эффекте множественного наследования. Предположим, что у нас есть C++ - класс *CA*:

```
class CA : public IX, public IY { ... }
```

Так как *CA* наследует и *IX*, и *IY*, то мы можем использовать указатель на *CA* везде, где можно использовать указатель на *IX* или *IY*. Указатель на *CA* можно передать функции, принимающей указатель на *IX* или *IY*, и функция будет работать правильно. Например:

```
void foo(IX* pIX);
void bar(IY* pIY);

int main()
{
 CA* pA = new CA;
 foo(pA);
 bar(pA);
 delete pA;
 return 0;
}
```

foo требуется указатель на указатель таблицы виртуальных функций *IX*, тогда как *bar* — указатель на указатель таблицы виртуальный функций *IY*. Содержимое таблиц виртуальных функций *IX* и *IY*, конечно же, разное. Мы не можем передать *bar* указатель vtbl *IX* и ожидать, что функция будет работать. Таким образом, компилятор не может передавать один и тот же указатель и *foo*, и *bar*, он должен модифицировать указатель на *CA* так, чтобы тот указывал на подходящий указатель виртуальной таблицы. На рис. 4.3 показан формат размещения объекта *CA* в памяти.

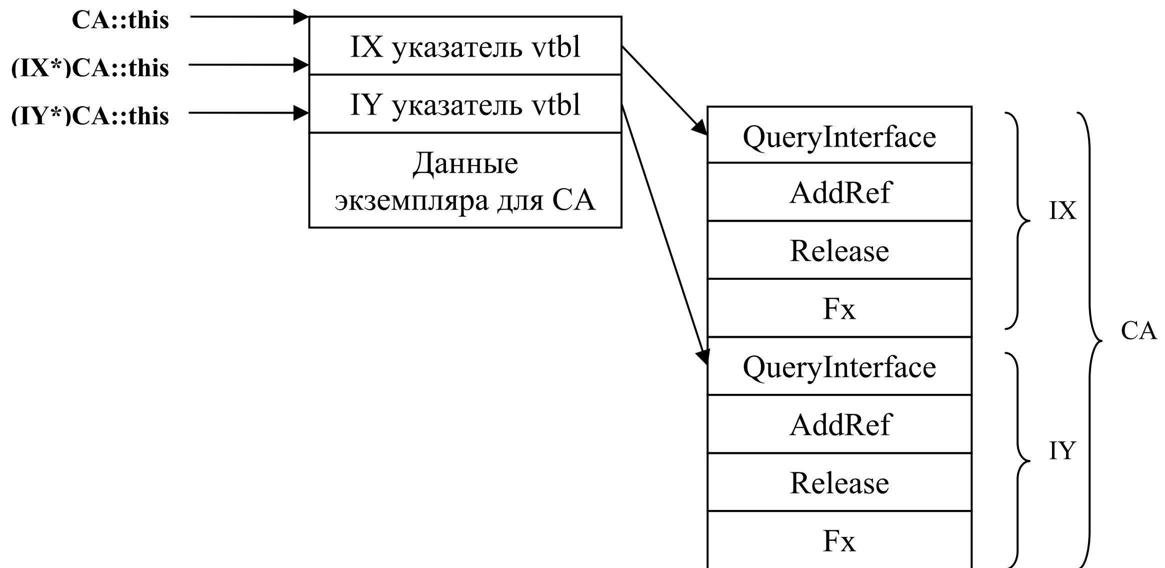


Рис. 4.3 *Формат памяти для класса CA, который множественно наследует IX и IY*

Из рис. 4.3 видно, что указатель *this* для *CA* указывает на указатель таблицы виртуальных функций *IX*. Таким образом, мы можем без изменения использовать для *CA* указатель *this* вместо указателя на *IX*. Однако очевидно, что указатель *this* для *CA* не указывает на указатель vtbl *IY*. Следовательно, указатель *this CA* надо модифицировать, прежде чем передавать функции, ожидающей указатель на *IY*. Для этого компилятор добавляет к указателю *this CA* смещение указателя vtbl *IY* (□*IY*). Приведенный ниже код:

```
IY* pC = pA;
```

компилятор транслирует во что-то вроде

```
IY* pC = (char*)pA + ΔIY;
```

Более подробную информацию Вы можете найти в разделе «Multiple Inheritance and Casting» книги Маргарет А. Эллис (Margaret A. Ellis) и Бьерна Страуструпа (Bjarne Strourstrup) *The Annotated C++ Reference Manual*. Компиляторы C++ не обязаны реализовывать vtbl при множественном наследовании именно так, как это показано на рис. 4.3

А теперь все вместе

Давайте соберем вместе все элементы и рассмотрим полный пример реализации и использования *QueryInterface*. В листинге 4.1 представлен полный текст этой простой программы. Программа состоит из трех частей.

В первой части объявляются интерфейсы *IX*, *IY* и *IZ*. Интерфейс *IUnknown* объявлен в заголовочном файле UNKNWN.H Win32 SDK.

Вторая часть — это реализация компонента. Класс *CA* реализует компонент, поддерживающий интерфейсы *IX* и *IY*. Реализация *QueryInterface* совпадает с той, что была представлена в предыдущем разделе книги. Функция *CreateInstance* определена после класса *CA*. Клиент использует ее, чтобы создать компонент, предоставляемый при помощи *CA*, и получить указатель на *IUnknown* этого компонента.

После *CreateInstance* следуют определения IID для интерфейсов. Как видно их этих определений, IID — весьма громоздкая структура (более подробно мы рассмотрим ее в гл. 8). Наш пример программы компонуется с UUID.LIB, чтобы получить определения для *IID_IUnknown* (т.е. IID для *IUnknown*).

Третья и последняя часть — функция *main*, которая выступает в качестве клиента.

IUNKNOWN.CPP

```
//
// IUnknown.cpp
// Чтобы скомпилировать: cl IUnknown.cpp UUID.lib
//

#include <iostream.h>
#include <objbase.h>

void trace(const char* msg) { cout << msg << endl; }

// Интерфейсы
interface IX : IUnknown
{
  virtual void __stdcall Fx() = 0;
};

interface IY : IUnknown
{
  virtual void __stdcall Fy() = 0;
};

interface IZ : IUnknown
{
  virtual void __stdcall Fz() = 0;
};

// Предварительные объявления GUID
extern const IID IID_IX;
extern const IID IID_IY;
extern const IID IID_IZ;

//
// Компонент
//
class CA : public IX, public IY
{
  // Реализация IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef() { return 0; }
  virtual ULONG __stdcall Release() { return 0; }
```

```
  // Реализация интерфейса IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }

  // Реализация интерфейса IY
  virtual void __stdcall Fy() { cout << "Fy" << endl; }
};

HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    trace("QueryInterface: Вернуть указатель на IUnknown");
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    trace("QueryInterface: Вернуть указатель на IX");
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    trace("QueryInterface: Вернуть указатель на IY");
    *ppv = static_cast<IY*>(this);
  }
  else
  {
    trace("QueryInterface: Интерфейс не поддерживается");
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef(); // См. гл. 5
  return S_OK;
}

//
// Функция создания
//

IUnknown* CreateInstance()
{
  IUnknown* pI = static_cast<IX*>(new CA);
  pI->AddRef();
  return pI;
}

//
// IID
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IX =
{0x32bb8320, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IY =
{0x32bb8321, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IZ =
{0x32bb8322, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

```
//
// Клиент
//
int main()
{
  HRESULT hr;
  trace("Клиент: Получить указатель на IUnknown");
  IUnknown* pIUnknown = CreateInstance();

  trace("Клиент: Получить указатель на IX");
  IX* pIX = NULL;
  hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: IX получен успешно");
    pIX->Fx(); // Использовать интерфейс IX
  }

  trace("Клиент: Получить указатель на IY");
  IY* pIY = NULL;
  hr = pIUnknown->QueryInterface(IID_IY, (void**)&pIY);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: IY получен успешно");
    pIY->Fy(); // Использовать интерфейс IY
  }

  trace("Клиент: Запросить неподдерживаемый интерфейс");
  IZ* pIZ = NULL;
  hr = pIUnknown->QueryInterface(IID_IZ, (void**)&pIZ);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: Интерфейс IZ получен успешно");
    pIZ->Fz();
  }
  else
  {
    trace("Клиент: Не могу получить интерфейс IZ");
  }

  trace("Клиент: Получить интерфейс IY через интерфейс IX");
  IY* pIYfromIX = NULL;
  hr = pIX->QueryInterface(IID_IY, (void**)&pIYfromIX);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: IY получен успешно");
    pIYfromIX->Fy();
  }

  trace("Клиент: Получить интерфейс IUnknown через IY");
  IUnknown* pIUnknownFromIY = NULL;
  hr = pIY->QueryInterface(IID_IUnknown, (void**)&pIUnknownFromIY);
  if (SUCCEEDED(hr))
  {
    cout << "Совпадают ли указатели на IUnknown? ";
    if (pIUnknownFromIY == pIUnknown)
    {
      cout << "Да, pIUnknownFromIY == pIUnknown" << endl;
    }
    else
    {
      cout << "Нет, pIUnknownFromIY != pIUnknown" << endl;
    }
  }
```

```
  // Удалить компонент
  delete pIUnknown;
  return 0;
}
```

Листинг 4.1 *Использование* QueryInterface

Эта программа выдает на экран следующее:

Клиент: Получить указатель на IUnknown
Клиент: Получить интерфейс IX
QueryInterface: Вернуть указатель на IX
Клиент: IX получен успешно
Fx
Клиент: Получить интерфейс IY
QueryInterface: Вернуть указатель на IY
Клиент: IY получен успешно
Fy
Клиент: Запросить неподдерживаемый интерфейс
QueryInterface: Интерфейс не поддерживается
Клиент: Не могу получить интерфейс IZ
Клиент: Получить интерфейс IY через IX
QueryInterface: Вернуть указатель на IY
Клиент: IY получен успешно
Fy
Клиент: Получить интерфейс IUnknown через IY
QueryInterface: Вернуть указатель на IUnknown
Совпадают ли указатели на IUnknown? Да, pIUnknownFromIY == pIUnknown

Клиент начинает с создания компонента при помощи *CreateInstance*. *CreateInstance* возвращает указатель на интерфейс *IUnknown* компонента. Клиент при помощи *QueryInterface* запрашивает через интерфейс *IUnknown* указатель на интерфейс *IX* компонента. Для проверки успешного окончания используется макрос SUCCEEDED.
Если указатель на *IX* получен успешно, то клиент с его помощью вызывает функцию этого интерфейса *Fx*.

Затем клиент использует указатель на *IUnknown*, чтобы получить указатель на интерфейс *IY*. В случае успеха клиент пользуется этим указателем. Поскольку класс *CA* реализует как *IX*, так и *IY*, *QueryInterface* успешно обрабатывает запросы на эти интерфейсы. Однако *CA* не реализует интерфейс *IZ*. Поэтому — когда клиент запрашивает этот интерфейс, *QueryInterface* возвращается код ошибки E_NOINTERFACE. Макрос SUCCEEDED возвращает FALSE, и *pIZ* не используется (для доступа к функциям-членам *IZ*).

Теперь мы дошли до по-настоящему интересных вещей. Клиент запрашивает указатель на интерфейс *IY* через указатель на интерфейс *IX*, *pIX*. Поскольку компонент поддерживает *IY*, этот запрос будет успешным, и клиент сможет использовать возвращенный указатель на интерфейс *IY* так же, как он использовал первый указатель.

Наконец, клиент запрашивает интерфейс *IUnknown* через указатель на *IY*. Поскольку все интерфейсы COM наследуют *IUnknown*, этот запрос должен быть успешным. Однако самое интересное, что возвращенный указатель на *IUnknown*, *pIUnknownFromIY*, совпадает с первым указателем на *IUnknown*, *pIUnknown*. Как мы увидим далее, это одно из требований COM: *QueryInterface* должна возвращать один и тот же указатель на все запросы к *IUnknown*.

Пример показывает, что при помощи *QueryInterface* можно получить любой из интерфейсов *CA* через любой другой. Это одно из важных правил реализации *QueryInterface*. Давайте более подробно рассмотрим его и другие правила.

Правила и соглашения *QueryInterface*

В этом разделе представлены некоторые правила, которым должны следовать все реализации *QueryInterface*. Если их выполнять, клиент сможет узнать о компоненте достаточно, чтобы (надеяться) управлять им и использовать его в своих целях. Без этих правил поведение *QueryInterface* было бы неопределенным, и писать программы было бы невозможно.

- Вы всегда получаете один и тот же *IUnknown*.
- Вы можете получить интерфейс снова, если смогли получить его раньше.
- Вы можете снова получить интерфейс, который у Вас уже есть.
- Вы всегда можете вернуться туда, откуда начали.
- Если Вы смогли попасть куда-то хоть откуда-нибудь, Вы можете попасть туда откуда угодно.

Теперь рассмотрим эти правила подробно.

Вы всегда получаете один и тот же *IUnknown*

У данного экземпляра компонента есть только один интерфейс *IUnknown*. Всегда, когда Вы запрашиваете у компонента *IUnknown* (не важно, через какой интерфейс), в ответ вы получите одно и то же значение указателя. Вы можете определить, указывают ли два интерфейса на один компонент, запросив у каждого из них *IUnknown* и сравнив результаты. Приведенная ниже функция *SameComponents* определяет, указывают ли *pIX* и *pIY* на интерфейсы одного компонента:

```
BOOL SameComponents(IX* pIX, IY* pIY)
{
  IUnknown* pI1 = NULL;
  IUnknown* pI2 = NULL;

  // Получить указатель на IUnknown через pIX
  pIX->QueryInterface(IID_IUnknown, (void**)&pI1);

  // Получить указатель на IUnknown через pIY
  pIY->QueryInterface(IID_IUnknown, (void**)&pI2);

  // Сравнить полученные указатели
  return pI1 == pI2;
}
```

Это важное правило. Без него нельзя было бы определить, указывают ли два интерфейса на один и тот же компонент.

Вы можете получить интерфейс снова, если смогли получить его раньше

Если *QueryInterface* однажды успешно обработала запрос на некоторый интерфейс, то все последующие запросы для того же компонента будут успешными. Если же запрос был неудачным, то для этого интерфейса *QueryInterface* всегда будет возвращать ошибку. Это правило относится только к конкретному экземпляру компонента. Ко вновь созданному экземпляру оно неприменимо.

Представьте себе, что произошло бы, если бы набор поддерживаемых интерфейсов мог изменяться со временем. Писать код клиента было бы крайне сложно. Когда клиент должен запрашивать интерфейсы у компонента? Как часто это делать? Что произойдет, если клиент не сможет получить интерфейс, который только что использовал? Без фиксированного набора интерфейсов клиент не мог бы сколько-нибудь уверенно определить возможности компонента.

Вы можете снова получить интерфейс, который у Вас уже есть

Если у Вас есть интерфейс *IX*, то Вы можете запроситьчерез него интерфейс *IX* и получите в ответ указатель на *IX*. Код выглядит так:

```
void f(IX* pIX)
{
  IX* pIX2 = NULL;

  // Запросить IX через IX
  HRESULT hr = pIX->QueryInterface(IID_IX, (void**)&pIX2);
  assert(SUCCEEDED(hr)); // Запрос должен быть успешным
}
```

Это правило звучит несколько странно. Зачем Вам интерфейс, который у Вас уже есть? Вспомните, однако, что все интерфейсы полиморфны относительно *IUnknown* и многим функциям передается указатель на *IUnknown*. У этих функций должна быть возможность использовать любой указатель на *IUnknown* и получить по нему любой другой интерфейс. Это иллюстрирует приведенный ниже пример:

```
void f(IUnknown* pI)
{
  HRESULT hr;
  IX* pIX = NULL;

  // Запросить IX через pI
  hr = pI->QueryInterface(IID_IX, (void**)&pIX);

  // Что-нибудь содержательное
}

void main()
{
  // Получаем откуда-то указатель на IX
  IX* pIX = GetIX();
  // Передаем его в функцию
  f(pIX);
}
```

Функция *f* сможет получить указатель на *IX* по переданному ей указателю, хотя последний и так указывает на *IX*.

Вы всегда можете вернуться туда, откуда начали

Если у Вас есть указатель на интерфейс *IX* и с его помощью Вы успешно получаете интерфейс *IY*, то можно получить «обратно» интерфейс *IX* через указатель на *IY*. Иными словами, независимо от того, какой интерфейс у Вас есть сейчас, можно снова получить интерфейс, с которого Вы начали. Это иллюстрирует следующий код:

```
void f(IX* pIX)
{
  HRESULT hr;
  IX* pIX2 = NULL;
  IY* pIY = NULL;

  // Получить IY через IX
  hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
  if (SUCCEEDED(hr))
  {
    // Получить IX через IY
    hr = pIY->QueryInterface(IID_IX, (void**)&pIX2);
    // QueryInterface должна отработать успешно
    assert(SUCCEEDED(hr));
  }
}
```

Если Вы смогли попасть куда-то хоть откуда-нибудь, Вы можете попасть туда откуда угодно

Если Вы можете получить у компонента некоторый интерфейс, то его можно получить с помощью любого из интерфейсов, поддерживаемых компонентом. Если можно получить интерфейс *IY* через *IX*, а *IZ* — через *IY*, то *IZ* можно получить и через *IX*. В программе это выглядит так:

```
void f(IX* pIX)
{
  HRESULT hr;
  IY* pIY = NULL;

  // Запросить IY у IX
  hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
  if (SUCCEEDED(hr))
  {
    IZ* pIZ = NULL;
    // Запросить IZ и IY
    hr = pIY->QueryInterface(IID_IZ, (void**)&pIZ);
    if (SUCCEEDED(hr))
    {
      // Запросить IZ у IX
      hr = pIX->QueryInterface(IID_IZ, (void**)&pIZ);
      // Это должно работать
      assert(SUCCEEDED(hr));
    }
  }
}
```

Это правило делает *QueryInterface* пригодной для использования. Вообразите, что стало бы, если бы получение указателя на интерфейс зависело от того, через какой интерфейс Вы делаете запрос. Вы редактируете код своего клиента, переставляя две функции местами, — и все перестает работать. Написать клиент для такого компонента было бы практически невозможно.

Общая задача всех приведенных правил — сделать использование *QueryInterface* простым, логичным, последовательным и определенным. По счастью, при реализации *QueryInterface* для компонента следовать правилам нетрудно. Если компоненты реализуют *QueryInterface* корректно, то клиенту не нужды беспокоиться об этих правилах. Пожалуйста, учтите, что простота реализации и использования *QueryInterface* не снижают значения этой функции. В COM нет ничего важнее, чем *QueryInterface*.

***QueryInterface* определяет компонент**

QueryInterface — это самая важная часть COM, поскольку она определяет компонент. Интерфейсы, поддерживаемые компонентом, — это те интерфейсы, указатели на которые возвращает *QueryInterface*. Их определяет реализация *QueryInterface*, а не заголовочный файл для класса C++, реализующего компонент. Компонент не определяется и иерархией наследования этого класса C++. Его определяет *исключительно* реализация *QueryInterface*.

Поскольку реализация *QueryInterface* для клиента невидима, то он не знает, какие интерфейсы поддерживаются. Единственный способ, которым клиент может об этом узнать, — запросить компонент. Такой порядок совершенно отличен от обычного в C++, где клиент имеет заголовочный файл класса и знает обо всех членах последнего. В некотором смысле концепция COM больше напоминает знакомство с человеком на вечеринке, чем при приеме на работу. Придя наниматься на работу, человек предоставляет Вам резюме, его характеризующее. Такое резюме похоже на объявление класса C++. На вечеринке же новый знакомый не вручает Вам никакой «сводки данных» о себе; чтобы узнать о нем, Вы задаете вопросы. Это больше похоже на практику COM.

Вы не можете воспользоваться всеми знаниями сразу

Первым вопросом, который я задал при изучении COM, был: «Почему я не могу сразу запросить у компонента все его интерфейсы?». Ответ в духе Дзен гласит: «Что станешь ты делать со списком интерфейсов, поддерживаемых компонентом?» Оказывается, это очень хороший ответ (хотя он и сформулирован как вопрос).

Допустим на мгновение, что клиент может запросить у компонента все поддерживаемые им интерфейсы. Допустим, что наш компонент поддерживает интерфейсы *IX* и *IY*, но клиент был написан раньше и ничего не знает об *IY*. Итак, он создает компонент и запрашиваетего интерфейсы. Компонент должен был бы вернуть *IX* и *IY*. Клиенту не известен интерфейс *IY*, поэтому он никак не может его использовать. Чтобы клиент мог сделать что-либо осмысленное с непонятным интерфейсом, он должен был бы прочитать документацию этого интерфейса и написать соответствующий код. При сегодняшнем уровне развития технологии это невозможно. Иными словами, компонент может поддерживать только те интерфейсы, которые известны его программисту.
Точно так же клиент может поддерживать только те интерфейсы, о которых знает *его* программист.

COM все же предоставляет средство, *библиотеки типа (type libraries)*, для определения интерфейсов, которые предоставляет компонент, во время выполнения. Хотя клиент может использовать библиотеку типа для определения параметров функций некоторого интерфейса, он по-прежнему не знает, как писать программы, использующие эти функции. Эта работа остается программисту. Библиотеки типа будут рассматриваться в гл. 12.

Во многих случаях клиенты могут использовать только компоненты, реализующие определенный набор интерфейсов. Создавать компонент, запрашивать у него интерфейсы по одному и в конце концов выяснить, что он не поддерживает один из нужных, — это пустая трата времени. Чтобы ее избежать, можно определить объединенный набор интерфейсов как *категорию компонентов (component category)*. Затем компоненты могут опубликовать сведения о том, принадлежат ли они к некоторой категории. Эту информацию клиенты могут получить, не создавая компонентов; подробнее об этом будет рассказано в гл. 7.

Работа с новыми версиями компонентов

Как Вы уже знаете, интерфейсы COM неизменны. После того, как интерфейс опубликован и используется каким-либо клиентом, он никогда не меняется. Но что именно я имею в виду, когда говорю, что интерфейсы остаются теми же? У каждого интерфейса имеется уникальный идентификатор интерфейса (IID). Вместо того, чтобы изменять интерфейс фактически нужно создать новый, с новым IID. Если *QueryInterface* получает запрос состарым IID, она возвращает старый интерфейс. Если же *QueryInterface* получает запрос с новым IID, то возвращает новый интерфейс. С точки зрения *QueryInterface* IID *и есть интерфейс*.

QueryMultipleInterfaces

В распределенной COM (DCOM) определен новый интерфейс *ImultiQI*. Он имеет единственную функцию-член — *QueryMultipleInterfaces*. Эта функция позволяет клиенту запросить у компонента несколько интерфейсов за один вызов. Один вызов *QueryMultipleInterfaces* заменяет несколько циклов «запрос-ответ» по сети, что повышает производительность.

Итак, интерфейс, соответствующие данному IID, неизменен. Новый интерфейс может наследовать старый или быть совершенно другим. На существующие клиенты это не влияет, так как старый интерфейс не меняется. Новые же клиенты могут использовать как старые, так и новые компоненты, поскольку могут запрашивать как старый, так и новый интерфейс.

Основная сила этого метода работы с версиями в его прозрачности. Клиенту не требуется выполнять никаких действий, чтобы проверить, имеет ли он дело с корректной версией интерфейса. Если клиент не может найти интерфейс, то этот интерфейс некорректен. Существо интерфейса полностью связано с его версией. Если изменяется версия, то изменяется и интерфейс. Здесь не может возникнуть никакой путаницы.

В качестве примера предположим, что у нас есть программа моделирования полета, названная Pilot, которая использует средства разных поставщиков для моделирования летательных аппаратов. Для того, чтобы работать с Pilot, компонент-«летательный аппарат» должен реализовывать интерфейс *IFly*. Пусть у одного из поставщиков имеется такой компонент, называемый Bronco и поддерживающий интерфейс *IFly*. Мы решаем модернизировать Pilot и выпускаем новую версию, FastPilot. FastPilot расширяет набор «поведений» самолета при помощи интерфейса *IFastFly*, в дополнение к *IFly.* Компания, продающая Bronco, добавляет интерфейс *IFastFly* и создает FastBronco.

FastPilot по-прежнему поддерживает *IFly*, поэтому, если у пользователя есть копия Bronco, то FastPilot по-прежнему может ее использовать. FastPilot будет сначала запрашивать у компонента *IFlyFast*, а если компонент его не поддерживает, — *IFly*. FastBronco по-прежнему поддерживает *IFly*, так что если у кого-то есть старый Pilot, то FastBronco будет работать и с ним. На рис. 4.4 возможные взаимосвязи представлены графически.

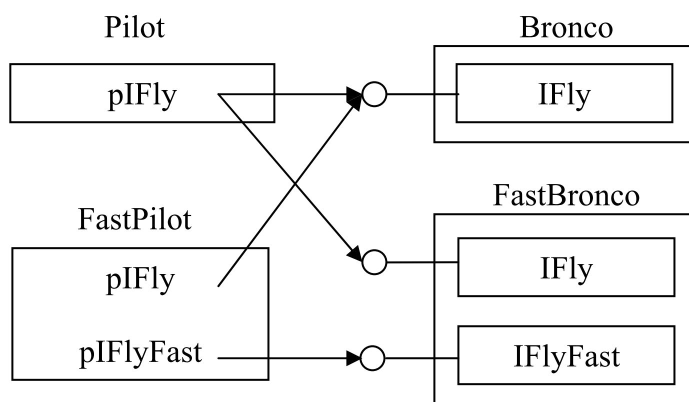


Рис. 4.4 *Различные комбинации старых и новых версий клиентов и компонентов*

В результате клиент и компонент смогут работать в любом сочетании. В некоторых случаях новый компонент или клиент не могут поддерживать обратную совместимость, поскольку та сложна или слишком медленно работает. Однако и тогда обработка версий в COM не теряет своей силы. Как и раньше, IID интерфейса определяет его версию. Всякий раз, когда клиент получает интерфейс, он получает корректную версию, поскольку другая версия — это другой интерфейс с другим IID.

Когда нужно создавать новую версию

Для того, чтобы механизм версий COM работал, программисты должны строго следить за присвоением новым версиям существующих интерфейсов новых IID. Вы обязаны создать новый интерфейс с новым IID, если изменили хотя бы одну из приведенных ниже характеристик:

- число функций в интерфейсе;
- порядок следования функций в интерфейсе;
- число параметров функции;
- типы параметров функции;
- возможные возвращаемые функцией значения;
- типы возвращаемых значений;
- смысл параметров функции;
- смысл функций интерфейса.

Вообще говоря, любое изменение, которое может нарушить работу любого из существующих клиентов, требует нового интерфейса. (Конечно, если Вы контролируете и клиент, и компонент, степеней свободы больше.)

Имена версий интерфейсов

Если Вы создаете новую версию интерфейса, следует изменить и его имя. Стандартное соглашение COM на этот счет заключается в добавлении номера к концу имени. Согласно ему, *IFly* становится *IFly2*, а не *IFastFly*. Конечно, свои собственные интерфейсы Вы можете называть как угодно. Если же интерфейсы принадлежат кому-либо другому, то следует запросить у него разрешение, прежде чем создавать новую версию или присваивать новое имя.

Неявные соглашения

Неизменность имен и параметров функций еще не гарантирует, что модификация компонента не повредит клиенту. Клиент использует функции интерфейса определенным образом или в определенной последовательности. Если реализация компонента изменяется так, что этот образ или последовательность действий не срабатывают, то работа клиента нарушится.

Это можно представить себе при помощи аналогии. Юридические договоры должны ясно и четко указывать обязанности сторон. Однако, позже, сколь бы коротким и простым ни был договор, в нем всегда найдется набранное мелким шрифтом. И это обязательно будет что-то, что Вы не считали важным, когда подписывали бумаги, — но что теперь может обойтись в тысячи долларов. Размер шрифта не имеет значения, юридическая сила зависит не от него.

Интерфейс — своего рода форма договора между клиентом и компонентом. Как и во всех договорах, здесь есть кое-что «мелким шрифтом». В случае интерфейсов это способ их использования. Способ, которым клиент использует функции интерфейса, составляет предмет договора с компонентом, реализующим интерфейс. Если компонент изменяет реализацию интерфейса, он должен гарантировать, что клиент сможет пользоваться функциями прежним способом. В противном случае клиент не будет работать, и его придется перекомпилировать. Пусть, например, клиент вызывает функции *Foo1*, *Foo2* и *Foo3* в этом порядке. Если компонент изменится так, что первой нужно будет вызывать *Foo3*, он нарушит неявное соглашение, определяющее способ и последовательность использования функций интерфейса.

Все интерфейсы «заключают» неявные соглашения. Это становится проблемой, только если мы хотим реализовать интерфейс способом, не совместимым с принятым порядком. Чтобы избежать нарушения неявного соглашения, у Вас есть два варианта. Первый заключается в том, чтобы сделать интерфейс работоспособным независимо от последовательности и способа вызова его функций-членов. Второй вариант — заставить всех клиентов использовать интерфейс одинаково и документировать этот порядок. Теперь, если компонент изменяется и нарушает работу клиента, он разрывает явный договор, а не неявный. Оба решения требуют огромной предусмотрительности и тщательного планирования.

5. Подсчет ссылок

В детстве я хотел стать пожарным. Романтика приключений и опасности привлекала меня, как и большинство мальчиков. Однако по-настоящему мне хотелось быть пожарным не потому. Дело в том, что, во-первых,пожарные ездили, повиснув сзади на пожарной машине (из-за этого я еще хотел стать мусорщиком, но это другая история). Во-вторых, у пожарных была по-настоящему крутая экипировка: металлические каски, высокие ботинки, большие плащи и кислородные баллоны. Я тоже хотел носить все эти замечательные вещи. Пожарные, казалось, никогда не расставались с ними. Даже если они всего лишь снимали кошку с дерева, то все равно делали это в касках, плащах и высоких ботинках. Пожарного было видно издалека.

Класс С++ кое в чем напоминает пожарного. Заголовок сообщает всему миру, какие сервисы и функции предоставляет класс, — точно так же, как амуниция пожарных говорит об их профессии. Однако компоненты СОМ ведут себя совершенно иначе. Компонент СОМ гораздо более скрытен, чем пожарный или классС++. Клиент не может посмотреть на компонент и сразу увидеть, что тот реализует пожарного. Вместо этого он должен выспрашивать: «Есть ли у Вас кислородный баллон? А топор? Носите ли Вы водонепроницаемую одежду?»

На самом деле клиенту неважно, имеет ли он дело с настоящим компонентом-пожарным. Ему важно, что у компонента за «амуниция». Например, если клиент задает вопрос «Носите ли Вы водонепроницаемую одежду?», то его удовлетворят ответы не только пожарного, но и байдарочника, аквалангиста и лесоруба в непромокаемом плаще. На вопрос «Есть ли у Вас кислородный баллон?» утвердительно ответит пожарный и аквалангист. На следующий вопрос про топор положительный ответ даст уже только пожарный (или аквалангист-лесоруб, если такой найдется).

Изоляция клиента от подлинной сущности компонента делает его менее восприимчивым к изменениям последнего. Однако, поскольку клиент знает компонент только через интерфейсы, он не может прямо управлять временем жизни компонента как такового. В этой главе мы рассмотрим косвенное управление — реализованное как явное управление временем жизни отдельных интерфейсов.

Управление временем жизни

Давайте разберем, почему клиент не должен управлять временем жизни компонента напрямую. Предположим, что наш клиент обращается к тому же компоненту-пожарному. В разных местах кода клиента могут быть разбросаны вызовы этого компонента через различные интерфейсы. Одна часть клиента может дышать через кислородный баллон при посредстве интерфейса *IUseOxygen*, а другая — крушить дом топором при помощи *IUseAxe*. Клиент может закончить пользоваться *IUseAxe* раньше, чем *IUseOxygen*. Однако Вы вряд ли захотите удалить компонент из памяти, когда с одним интерфейсом уже закончена, а с другим еще продолжается. Определить момент, когда компонент можно безопасно удалить, сложно еще и потому, что мы не знаем, указывают ли два указателя на интерфейсы одного и того же компонента. Единственный способ узнать это — запросить *IUnknown* через оба интерфейса и сравнить результаты. По мере того, как программа усложняется, становится все труднее определить, когда можно «отпускать» компонент. Проще всего загрузить его и не выгружать до завершения приложения. Но такое решение не слишком эффективно.

Итак, наша стратегия будет такова: вместо того, чтобы удалять компоненты напрямую, мы будем сообщать компоненту, что нам нужен интерфейс или что мы закончили с ним работать. Мы точно знаем, когда начинаем использовать интерфейс, и знаем (обычно), когда перестаем его использовать. Однако, как уже ясно, мы можем

не знать, что закончили использовать компонент вообще. Поэтому имеет смысл ограничиться сообщением об окончании работы с данным интерфейсом — и пусть компонент сам отслеживает, когда мы перестаем пользоваться всеми интерфейсами.
Именно для реализации этой стратегии и предназначены еще две функции-члена *IUnknown* — *AddRef* и *Release*. В прошлой главе было дано определение интерфейса *IUnknown*:

```
interface IUnknown
{
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) = 0;
  virtual ULONG __stdcall AddRef() = 0;
  virtual ULONG __stdcall Release() = 0;
};
```

В этой главе мы рассмотрим, как *AddRef* и *Release* дают компоненту возможность контролировать время своей жизни, а клиенту — возможность иметь дело с интерфейсами. Мы начнем с обзора подсчета ссылок, а затем посмотрим, как клиент использует *AddRef* и *Release*. Познакомившись с использованием функций, мы реализуем их в компоненте. Наконец, мы обсудим, когда вызовы *AddRef* и *Release* можно опустить для повышения производительности и подытожим наше обсуждение набором правил.

Подсчет ссылок

AddRef и *Release* реализуют и технику управления памятью, известную как *подсчет ссылок (reference counting)*. Подсчет ссылок — простой и быстрый способ, позволяющий компонентам самим удалять себя. Компонент СОМ поддерживает *счетчик ссылок*. Когда клиент получает некоторый интерфейс, значение этого счетчика увеличивается на единицу. Когда клиент заканчивает работу с интерфейсом, значение на единицу уменьшается. Когда оно доходит до нуля, компонент удаляет себя из памяти. Клиент также увеличивает счетчик ссылок, когда создает новую ссылку на уже имеющийся у него интерфейс. Как Вы, вероятно, догадались, увеличивается счетчик вызовом *AddRef*, а уменьшается — вызовом *Release*. Для того, чтобы пользоваться подсчетом ссылок, необходимо знать лишь три простых правила:

1. **Вызывайте *AddRef* перед возвратом.** Функции, возвращающие интерфейсы, перед возвратом всегда должны вызывать *AddRef* для соответствующего указателя. Это также относится к *QueryInterface* и функции *CreateInstance*. Таким образом, Вам не нужно вызывать *AddRef* в своей программе после получения (от функции) указателя на интерфейс.
2. **По завершении работы вызывайте *Release*.** Когда Вы закончили работу с интерфейсом, следует вызвать для него *Release*.
3. **Вызывайте *AddRef* после присваивания.** Когда бы Вы ни присваивали один указатель на интерфейс другому, вызывайте *AddRef*. Иными словами: следует увеличить счетчик ссылок каждый раз, когда создается новая ссылка на данный интерфейс.

Вот три простых правила подсчета ссылок. Теперь рассмотрим несколько примеров. Для начала — простой пример «на первые два правила». Приведенный ниже фрагмент кода создает компонент и получает указатель на интерфейс *IX*. Мы не вызываем *AddRef*, так как за нас это делают *CreateInstance* и *QueryInterface*. Однако мы вызываем *Release* как для интерфейса *IUnknown*, возвращенного *CreateInstance*, так и для интерфейса *IX*, возвращенного *QueryInterface*.

```
// Создать компонент
IUnknown* pIUnknown = CreateInstance();

// Получить интерфейс IX
IX* pIX = NULL;
HRESULT hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
if (SUCCEEDED(hr))
{
  pIX->Fx(); // Использовать интерфейс IX
  pIX->Release(); // Завершить работу с IX
}

pIUnknown->Release(); // Завершить работу с IUnknown
```

В приведенном выше примере мы фактически закончили работать с *IUnknown* сразу же после вызова *QueryInterface*, так что его можно освободить раньше.

```
// Создать компонент
IUnknown* pIUnknown = CreateInstance();

// Получить интерфейс IX
IX* pIX = NULL;
HRESULT hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);

// Завершить работу с IUnknown
pIUnknown->Release();

// Использовать IX, если он был получен успешно
if (SUCCEEDED(hr))
{
  pIX->Fx(); // Использовать интерфейс IX
  pIX->Release(); // Завершить работу с IX
}
```

Легко забыть, что всякий раз, когда Вы копируете указатель на интерфейс, надо увеличить его счетчик ссылок. В приведенном далее фрагменте кода делается еще одна ссылка на интерфейс *IX*. В общем случае необходимо увеличивать счетчик ссылок *всякий раз*, когда создается копия указателя на интерфейс, о чем говорит приведенное выше правило 3.

```
// Создать компонент
IUnknown* pIUnknown = CreateInstance();
IX* pIX = NULL;
HRESULT hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
pIUnknown->Release();
if (SUCCEEDED(hr))
{
  pIX->Fx(); // Использовать интерфейс IX
  IX* pIX2 = pIX; // Создать копию pIX
  pIX2->AddRef(); // Увеличить счетчик ссылок
  pIX2->Fx(); // Что-то делать при помощи pIX2
  pIX2->Release(); // Завершить работу с pIX2
  pIX->Release(); // Завершить работу с pIX
}
```

Первая Ваша реакция на показанный выше код могла быть такой: «Обязательно ли нужно вызывать в этом примере *AddRef* и *Release* для *pIX2*?» либо «Как я запомню, что всякий раз при копировании указателя нужно вызывать *AddRef* и *Release*?» У некоторых оба эти вопроса возникают одновременно. Ответ на первый вопрос — нет. В данном

примере *AddRef* и *Release* для *pIX2* вызывать необязательно. В простых случаях, вроде этого, легко заметить, что увеличение и уменьшение счетчика ссылок для *pIX2* излишне, поскольку время жизни *pIX2* совпадает со временем жизни *pIX*. Правила оптимизации подсчета ссылок будут рассмотрены ниже в этой главе. Однако в этом случае следует вызывать *AddRef* всякий раз, когда порождается новое имя для указателя на интерфейс. В реальных, не простых случаях гораздо сложнее понять, отсутствует ли вызов *AddRef* и *Release* по ошибке или вследствие оптимизации. Как человек, которому приходилось целыми днями искать, почему ссылки подсчитываются неправильно, могу Вас уверить, что решать такие проблемы нелегко.

Однако, как мы увидим в гл. 11, классы smart-указателей позволяют полностью инкапсулировать подсчет ссылок.

Еще раз: клиент сообщает компоненту о своем желании использовать интерфейс, когда вызывается *QueryInterface*. Как мы видели выше, *QueryInterface* вызывает *AddRef* для запрашиваемого интерфейса. Когда клиент заканчивает работу с интерфейсом, он вызывает для этого интерфейса *Release*. Компонент остается в памяти, ожидая, пока счетчик ссылок не станет равен 0. Когда счетчик становится нулем, компонент сам себя удаляет.

Подсчет ссылок на отдельные интерфейсы

Я должен отметить одну тонкую деталь. С точки зрения клиента, подсчет ссылок ведется на уровне интерфейсов, а не на уровне компонентов. Помните пожарного с амуницией? Клиент не может видеть все целиком, он видит только интерфейсы. Таким образом, клиент считает, что у каждого интерфейса — свой счетчик ссылок.

Итак, хотя с точки зрения клиентов подсчет ссылок осуществляется для интерфейсов, а не для компонентов (см. рис. 5.1), для реализации компонента это не имеет значения. Компонент может поддерживать отдельные счетчики для каждого из интерфейсов, а может и иметь один общий счетчик. Реализация не имеет значения до тех пор, пока клиент убежден, что подсчет ссылок ведется для самих интерфейсов. Поскольку компонент может реализовывать подсчет для каждого интерфейса, постольку клиент не должен предполагать обратного.

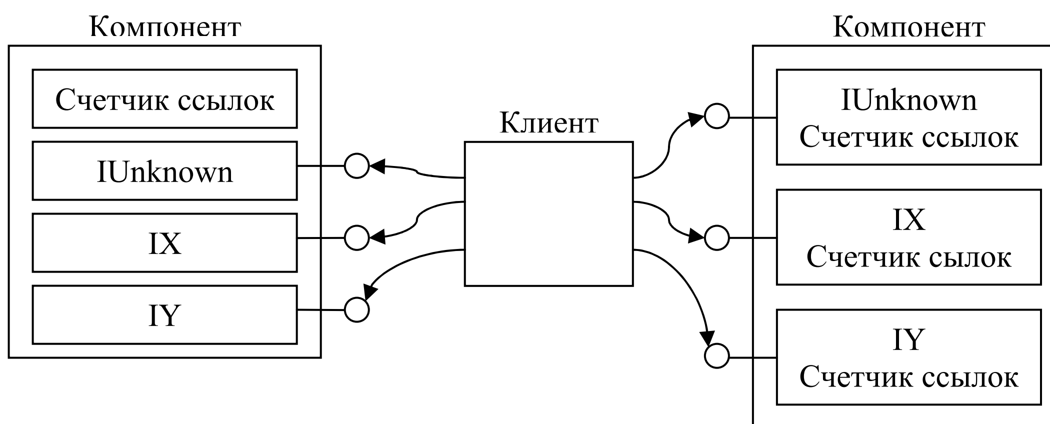


Рис. 5.1 *Программист компонента может использовать один счетчик ссылок для всего компонента либо отдельные счетчики для каждого интерфейса*

Что означает для клиента подсчет ссылок для каждого интерфейса в отдельности? Он означает, что клиент должен вызывать *AddRef* именно для того указателя, с которым собирается работать, а не для какого-нибудь другого. Клиент также должен вызывать *Release* именно для того указателя, с которым закончил работу.

Например, не делайте так:

```
IUnknown* pIUnknown = CreateInstance();
IX* pIX = NULL;
pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
pIX->Fx();
IX* pIX2 = pIX;
pIUnknown->AddRef(); // Должно быть pIX2->AddRef();
pIX2->Fx();
pIX2->Release();
pIUnknown->Release(); // Должно быть pIX->Release();
pIUnknown->Release();
```

В приведенном фрагменте предполагается, что можно вызывать *AddRef* и *Release* через указатель на *IUnknown*, как если бы это был указатель на *IX*. В зависимости от реализации такой код может создавать проблемы. Зачем программисту компонента может потребоваться реализовывать подсчет ссылок для каждого интерфейса, а не для всего компонента в целом? По двум основным причинам: для упрощения отладки и для поддержки выделения ресурсов по требованию.

Отладка

Предположим, что Вы забыли вызвать *Release* для некоторого из интерфейсов компонента; забыть это легко. Компонент никогда не освободится, так как *delete* вызывается, только когда счетчик ссылок становится равным нулю. Проблема в том, чтобы найти, где и когда надо было освободить интерфейс; это может оказаться очень трудно. Поиски ошибок еще более усложняются, если у компонента один общий счетчик ссылок. В этом случае придется проверять все случаи использования всех предоставляемых компонентов интерфейсов. Если же компонент поддерживает для каждого интерфейса отдельный счетчик ссылок, то поиск ограничивается местами использования нужного интерфейса. Иногда это экономит массу времени.

Выделение ресурсов по требованию

Реализация интерфейса может потребовать большего объема памятиили других ресурсов. Достаточно просто реализовать *QueryInterface* так, чтобы память выделялась в момент запроса интерфейса. Однако, если имеется только один счетчик ссылок на весь компонент, нельзя определить, когда можно безопасно освободить память, связанную с данным интерфейсом. Использование отдельных счетчиков упрощает задачу.

Другой, и в большинстве случаев лучший, вариант — реализовать «ресурсоемкий» интерфейс в отдельном компоненте и передавать клиенту интерфейс последнего. Эта техника, называемая *агрегированием (aggregation)*, будет продемонстрирована в гл. 9.

Для того чтобы примеры в книге были проще, мы будем использовать в компонентах общий счетчик ссылок.
Теперь давайте рассмотрим, как реализовать подсчет ссылок.

Реализация *AddRef* и *Release*

Реализация *AddRef* (и *Release*) относительно проста. В основном она сводится к операции увеличения (уменьшения) на единицу, как показано ниже.

```
ULONG __stdcall AddRef()
{
  return ++m_cRef;
}

ULONG __stdcall Release()
{
  if (--m_cRef == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}
```

AddRef увеличивает значение переменной *m_cRef*, счетчика ссылок. *Release* уменьшает *m_cRef* и удаляет компонент, если значение переменной становится равным нулю. Во многих случаях Вы можете встретить реализацию *AddRef* и *Release* при помощи функций Win32 *InterlockedIncrement* и *InterlockedDecrement*. Эти функции гарантируют, что значение переменной изменяет в каждый момент времени только один поток управления. В зависимости от потоковой модели, используемой Вашим объектом СОМ, параллельные потоки могут создавать проблемы. Вопросы, связанные с потоками, будут рассмотрены в гл. 13.

```
ULONG __stdcall AddRef()
{
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}
```

Вам также следует знать, что возвращаемые *AddRef* и *Release* значения не имеют смысла и использовать их можно только для отладки. Клиент не должен полагаться на то, что эти значения как-то связаны с числом ссылок на компонент или его интерфейсы.

Если Вы внимательно читали код гл. 4, то заметили, что я уже использовал *AddRef* в двух местах — в *QueryInterface* и *CreateInstance*.

```
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    *ppv = static_cast<IY*>(this);
  }
  else
```

```
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  static_cast<IUnknown*>(*ppv)->AddRef(); // См. гл. 5
  return S_OK;
}

IUnknown* CreateInstance()
{
  IUnknown* pI = static_cast<IX*>(new CA);
  pI->AddRef();
  return pI;
}
```

Всякий раз, создавая компонент, Вы создаете и ссылку на него. Таким образом, компонент в момент создания должен увеличивать счетчик ссылок, прежде чем возвратить клиенту указатель. Это освобождает программиста от необходимостипомнить, что после *CreateInstance* и *QueryInterface* надо вызывать *AddRef*.

В некоторых случаях вызовы *AddRef* и *Release* можно опустить. Однако, прежде чем мы избавимся от некоторых из них, давайте рассмотрим листинг 5.1, который показывает все изложенное выше на примере.

REFCOUNT.CPP

```
//
// RefCount.cpp
// Копиляция: cl RefCount.cpp UUID.lib
//

#include <iostream.h>
#include <objbase.h>

void trace(const char* msg) { cout << msg << endl; }

// Предварительные описания GUID
extern const IID IID_IX;
extern const IID IID_IY;
extern const IID IID_IZ;

// Интерфейсы
interface IX : IUnknown
{
  virtual void __stdcall Fx() = 0;
};

interface IY : IUnknown
{
  virtual void __stdcall Fy() = 0;
};

interface IZ : IUnknown
{
  virtual void __stdcall Fz() = 0;
};


//
// Компонент
//
```

```
class CA : public IX, public IY
{
  // Реализация IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // Реализация интерфейса IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }

  // Реализация интерфейса IY
  virtual void __stdcall Fy() { cout << "Fy" << endl; }

public:
  // Конструктор
  CA() : m_cRef(0) {}

  // Деструктор
  ~CA() { trace("CA: Ликвидировать себя"); }

private:
  long m_cRef;
};

HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    trace("CA QI: Возвратить указатель на IUnknown");
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    trace("CA QI: Возвратить указатель на IX");
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    trace("CA QI: Возвратить указатель на IY");
    *ppv = static_cast<IY*>(this);
  }
  else
  {
    trace("CA QI: Интерфейс не поддерживается");
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CA::AddRef()
{
  cout << "CA: AddRef = " << m_cRef+1 << endl;
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CA::Release()
{
  cout << "CA: Release = " << m_cRef-1 << endl;
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
```

```
  }
  return m_cRef;
}

//
// Функция создания
//

IUnknown* CreateInstance()
{
  IUnknown* pI = static_cast<IX*>(new CA);
  pI->AddRef();
  return pI;
}

//
// IID
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}

static const IID IID_IX =
{0x32bb8320, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}

static const IID IID_IY =
{0x32bb8321, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}

static const IID IID_IZ =
{0x32bb8322, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

//
// Клиент
//

int main()
{
  HRESULT hr;

  trace("Клиент: Получить указатель на IUnknown");
  IUnknown* pIUnknown = CreateInstance();

  trace("Клиент: Получить интерфейс IX");
  IX* pIX = NULL;
  hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: IX получен успешно");
    pIX->Fx(); // Использовать интерфейс IX
    pIX->Release();
  }

  trace("Клиент: Получить интерфейс IY");
  IY* pIY = NULL;
  hr = pIUnknown->QueryInterface(IID_IY, (void**)&pIY);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: IY получен успешно");
    pIY->Fy(); // Использовать интерфейс IY
    pIY->Release();
  }
```

```
  trace("Клиент: Запросить неподдерживаемый интерфейс");
  IZ* pIZ = NULL;
  hr = pIUnknown->QueryInterface(IID_IZ, (void**)&pIZ);
  if (SUCCEEDED(hr))
  {
    trace("Клиент: Интерфейс IZ получен успешно");
    pIZ->Fz();
    pIZ->Release();
  }
  else
  {
    trace("Клиент: Не могу получить интерфейс IZ");
  }

  trace("Клиент: Освободить интерфейс IUnknown");
  pIUnknown->Release();
  return 0;
}
```

Листинг 5.1 *Полный пример подсчета ссылок*

Эта программа выводит на экран следующее:

Клиент: Получить указатель на IUnknown
CA: AddRef = 1
Клиент: Получить указатель на IX
CA QI: Вернуть указатель на IX
CA: AddRef = 2
Клиент: IX получен успешно
Fx
CA: Release = 1
Клиент: Получить интерфейс IY
CA QI: Вернуть указатель на IY
CA: AddRef = 2
Клиент: IY получен успешно
Fy
CA: Release = 1
Клиент: Запросить неподдерживаемый интерфейс
CA QI: Интерфейс не поддерживается
Клиент: Не могу получить интерфейс IZ
Клиент: Освободить интерфейс IUnknown
CA: Release = 0
CA: Ликвидировать себя

Это та же программа, что и в примере гл. 4, но к ней добавлен подсчет ссылок. К компоненту добавлены реализации *AddRef* и *Release*. Единственное отличие в клиенте — добавлены вызовы *Release*, чтобы обозначить окончание работы с различными интерфейсами. Обратите также внимание, что клиент больше не использует оператор *delete*. В данном примере клиенту нет надобности в *AddRef*, так как эту функцию для соответствующих указателей вызывают *CreateInstance* и *QueryInterface*.

Когда подсчитывать ссылки

Теперь пора разобраться с тем, когда нужно вести подсчет ссылок. Мы увидим, что иногда можно безопасно опустить пары вызовов *AddRef*/*Release*, тем самым оптимизируя

код. Сочетая изложенные ранее принципы с новыми навыками оптимизации, мы определим некоторые общие правила подсчета ссылок.

Оптимизация подсчета ссылок

Не так давно мы задавали себе вопрос, нужно ли при копировании указателя интерфейса всякий раз увеличивать счетчик ссылок. Этот вопрос появился при рассмотрении кода, похожего на приведенный ниже:

```
HRESULT hr;

IUnknown* pIUnknown = CreateInstance();
IX* pIX = NULL;
hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
pIUnknown->Release();
if (SUCCEEDED(hr))
{
  IX* pIX2 = pIX; // Скопировать pIX
    // Время жизни pIX2 «вложено» во время существования pIX
    pIX2->AddRef(); // Увеличить счетчик ссылок
    pIX->Fx(); // Использовать интерфейс IX
    pIX2->Fx(); // Сделать что-нибудь при помощи pIX2
    pIX2->Release(); // Конец работы с pIX2
  pIX->Release(); // Конец работы с IX
  // А также конец работы с компонентом
}
```

Представленный фрагмент не выгружает компонент до тех пор, пока клиент не освободит *pIX*. Клиент не освобождает *pIX* до тех пор, пока не закончит работу как с *pIX*, так и с *pIX2*. Поскольку компонент не выгружается, пока не освобожден *pIX*, постольку он гарантированно остается в памяти на протяжении всей жизни *pIX2*. Таким образом, нам на самом деле нет необходимости вызывать *AddRef* и *Release* для *pIX2*, поэтому две строки кода, выделенные полужирным шрифтом, можно сократить.

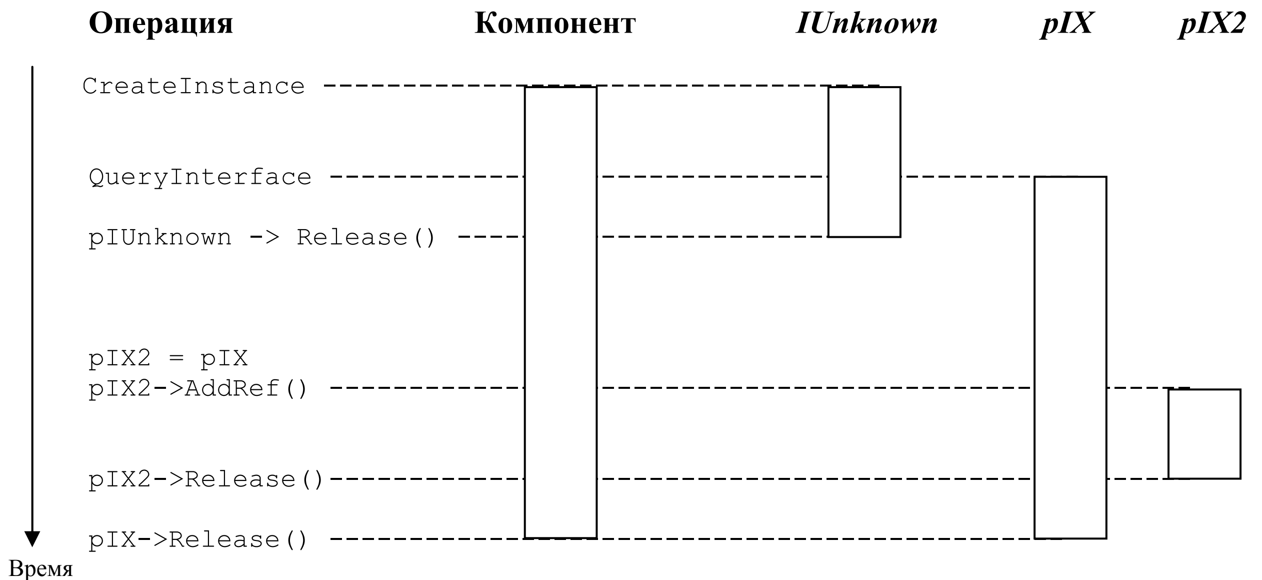
Подсчет ссылок для *pIX* — это все, что необходимо для удерживания компонента в памяти. Принципиально то, что время жизни *pIX2* содержится внутри времени существования *pIX*. Чтобы подчеркнуть это, я увеличил отступы для строк, где используется *pIX2*. На рис. 5.2 вложение времени существования *pIX* и *pIX2* показано в виде графика. Здесь столбиками обозначены времена жизни различных интерфейсов и время жизни самого компонента. Ось времени направлена сверху вниз. Операции, оказывающие воздействие на продолжительность жизни, перечислены в левой части рисунка. Горизонтальные линии показывают, как эти операции начинают или завершают период существования интерфейсов.

Из рис. 5.2 легко видеть, что жизни *pIX2* начинается после начала жизни *pIX* и заканчивается до окончания жизни *pIX*. Таким образом, счетчик ссылок *pIX* будет сохранять компонент в памяти все время жизни *pIX2*. Если бы жизнь *pIX2* не содержалась внутри жизни *pIX*, но перекрывалась с нею, то для *pIX2* потребовалось бы подсчитывать ссылки. Например, в следующем фрагменте кода жизни *pIX2* и *pIX* перекрываются:

```
HRESULT hr;

IUnknown* pIUnknown = CreateInstance();
IX* pIX = NULL;
hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
pIUnknown->Release();
if (SUCCEEDED(hr))
{
  IX* pIX2 = pIX; // Скопировать pIX
```

```
  pIX2->AddRef(); // Начало жизни pIX2
  pIX->Fx();
  pIX->Release(); // Конец жизни IX
  pIX2->Fx();
  pIX2->Release(); // Конец жизни pIX2
  // А также конец работы с компонентом
}
```



Столбиками показаны времена жизни различных элементов

Рис. 5.2 *Вложенность времен жизни указателей на интерфейсы. Ссылки для указателя со вложенным временем жизни подсчитывать не требуется.*

В этом примере мы обязаны вызывать *AddRef* для *pIX2*, так как *pIX2* освобождается после освобождения *pIX*. Графически это представлено на рис. 5.3.

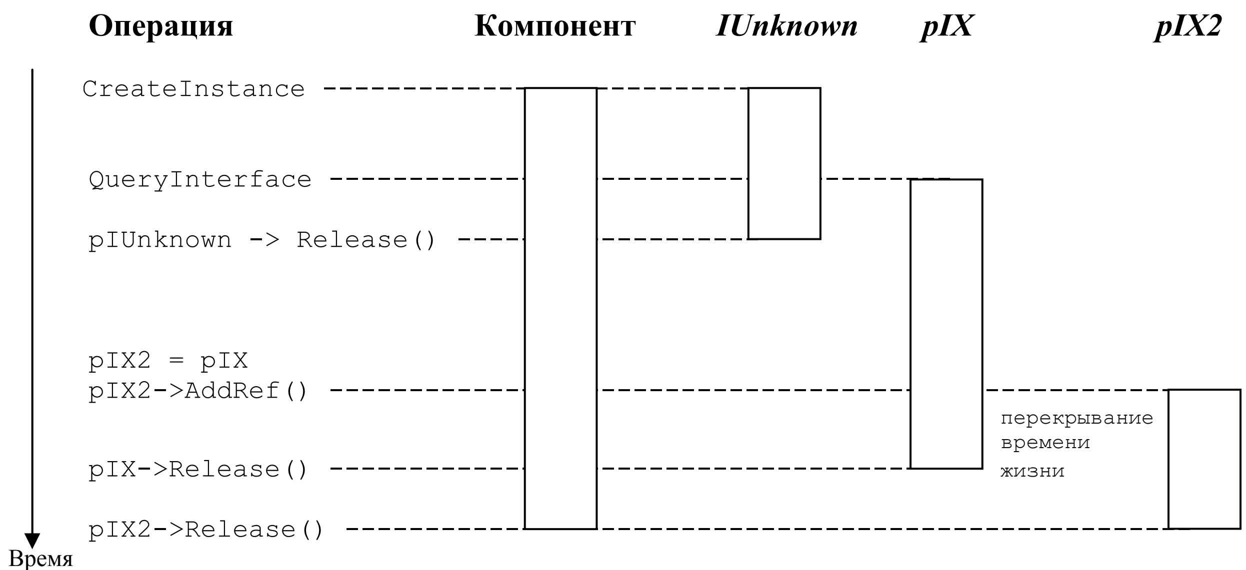


Рис. 5.3 *Перекрывание времен жизни указателей на интерфейсы. Здесь надо подсчитывать ссылки на оба интерфейса.*

В таких простых примерах легко определить, нужно ли подсчитывать ссылки. Однако достаточно лишь немного приблизиться к реальности, как идентифицировать вложенность времен жизни будет затруднительно. Тем не менее, иногда соотношение времен жизни по-прежнему очевидно. Один такой случай — функции. Для нижеследующего кода очевидно, что время работы *foo* содержится внутри времени жизни *pIX*. Таким образом, нет необходимости вызывать *AddRef* и *Release* для передаваемых в функцию указателей на интерфейсы.

```
void foo(IX* pIX2)
{
  pIX2->Fx(); // Использование интерфейса IX
}

void main()
{
  HRESULT hr;
  IUnknown* pIUnknown = CreateInstance();
  IX* pIX = NULL;
  hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
  pIUnknown->Release();
  if (SUCCEEDED(hr))
  {
    foo(pIX); // Передать pIX процедуре
    pIX->Release(); // Завершить работу с IX
    // А также и с компонентом
  }
}
```

Внутри функции незачем подсчитывать ссылки для указателей на интерфейсы, хранящиеся в локальных переменных. Время жизни локальной переменной совпадает со временем работы функции, т.е. содержится внутри времени жизни вызывающей программы. Однако подсчет ссылок необходим при всяком копировании указателя в глобальную переменную или из нее — глобальная переменная может освободиться в любой момент и в любой функции.

Оптимизация подсчета ссылок основана на определении указателей на интерфейс, чьи времена жизни вложены во времена жизни других ссылок на тот же интерфейс. Искать такие вложения в непростом коде бывает сложно. Однако правила, представленные в следующем разделе, учитывают некоторые типичные случаи, когда пары *AddRef* / *Release* можно опустить без большой опасности внести в программу ошибку.

Правила подсчета ссылок

Эти правила объединяют идеи оптимизации из предыдущего раздела с правилами подсчета ссылок, приведенными в начале главы. Читая их, помните, что клиент должен работать с каждым интерфейсом так, как если бы у того был отдельный счетчик ссылок. Следовательно, клиент должен выполнять подсчет ссылок для разных указателей на интерфейсы, хотя бы их времена жизни и были вложенными.

Правило для выходных параметров

Выходной параметр (out parameter) — это параметр функции, в котором вызывающей программе возвращается некоторое значение. Функция устанавливает это значение; первоначальное, заданное вызывающей программой значение не используется. Выходные параметры служат той же цели, что и возвращаемые значения функции.

Пример выходного значения параметра — второй параметр функции *QueryInterface*.

```
HRESULT QueryInterface(const IID&, void**);
```

Любая функция, возвращающая указатель на интерфейс через выходной параметр или как свое собственное возвращаемое значение, должна вызывать *AddRef* для этого указателя. Это то же самое правило, что и «Вызывайте *AddRef* перед возвратом» из начала главы, но сформулировано оно по-другому. *QueryInterface* следует этому правилу, вызывая *AddRef* для возвращаемого ею указателя на интерфейс. Наша функция создания компонентов *CreateInstance* также следует ему.

Правило для входных параметров

Входной параметр (in parameter) — это параметр, через который функции передается некоторое значение. Функция использует это значение, но не изменяет его и ничего не возвращает в нем вызывающей программе. В C++ такие параметры представляются константами или передаваемымипо значению аргументами функции. Ниже указатель на интерфейс передается как входной параметр:

```
void foo(IX* pIX)
{
  pIX->Fx();
}
```

Указатель на интерфейс, переданный в функцию, не требует обращений к *AddRef* и *Release*, так как время жизни функции всегда вложено во время жизни вызывающей программы. Это правило легко запомнить, если попробовать мысленно подставить код функции в точку ее вызова. Возьмем в качестве примера следующий фрагмент:

```
IX* pIX = CreateInstance(); // Автоматический вызов AddRef
foo(IX);
pIX->Release();
```

В варианте с «развернутым» кодом *foo* этот фрагмент имел бы вид:

```
IX* pIX = CreateInstance(); // Автоматический вызов AddRef
// foo(pIX);
pIX->Fx(); // Подстановка функции foo
pIX->Release();
```

После установки *foo* становится очевидно, что время ее жизни вложено во время жизни вызывающей программы.

Правило для параметров типа вход-выход

Параметр типа вход-выход (in-out parameter) может одновременно быть и входным, и выходным. Функция использует переданное ей значение такого параметра, затем изменяет его и возвращает вызывающей программе.

Функция обязана вызвать *Release* для указателя на интерфейс, переданного ей как произвольный параметр, прежде чем записать на его место новый указатель. Перед возвратом в вызывающую программу функция также должна вызвать *AddRef* для нового значения параметра.

```
void ExchangeForChangedPtr(int i, IX** ppIX)
{
  (**ppIX)->Fx(); // Делаем что-нибудь с входным параметром
  (**ppIX)->Release(); // Освобождаем входной параметр
  *ppIX = g_Cache[i]; // Выбираем указатель из кэша
  (**ppIX)->AddRef(); // Вызываем для него AddRef
  (**ppIX)->Fx(); // Делаем что-нибудь с выходным параметром
}
```

Правило для локальных переменных

Локальные копии указателей на интерфейсы, конечно, существуют только во время выполнения функций и нетребуют пар *AddRef / Release*. Это правило непосредственно вытекает из правила для входных параметров. В приведенном далее примере *pIX2* гарантированно будет существовать только во время выполнения функции *foo*.
Таким образом, его существование вложено во время жизни указателя *pIX*, переданного как входной параметр, — так что вызывать *AddRef* или *Release* для *pIX2* не нужно.

```
void foo(IX* pIX)
{
  IX* pIX2 = pIX;
  pIX2->Fx();
}
```

Правило для глобальных переменных

Если указатель на интерфейс сохраняется в глобальной переменной, то прежде чем передавать управление другой функции, необходимо вызвать *AddRef*. Поскольку переменная является глобальной, любая функция может вызвать *Release* и закончить жизнь этого указателя. Указатели на интерфейсы, сохраняемые в переменных-членах, должны обрабатываться аналогично. Любая функция-член класса может изменить состояние такого указателя.

Правило для сомнительных случаев

Всякий раз, когда у Вас возникает сомнение, вставляйте пару *AddRef / Release*. Ее отсутствие редко дает значительный выигрыш в производительности или экономию памяти, зато легко может привести к созданию компонента, который никогда не удаляется из памяти. Кроме того, Вы можете потратить много времени на поиск ошибки, вызванной неправильным подсчетом ссылок. Как обычно, с помощью профилировщика можно определить, дает ли оптимизация существенный выигрыш. Помимо этого, если Вы все же решили применитьоптимизацию, обязательно пометьте указатель, для которого не выполняется подсчет ссылок, соответствующимкомментарием. Другому программисту, модифицирующему Ваш код, очень легко запутаться во временах жизни и нарушить правильность оптимизированного подсчета ссылок.

Пропущенный вызов *Release* труднее обнаружить, чем отсутствие вызова *AddRef*. Программисты на C++ легко могут забыть вызвать *Release* или, еще хуже, попытаться использовать *delete* вместо *Release*. В гл. 11 показано, как smart-указатели могут полностью инкапсулировать подсчет ссылок.

Амуниция пожарного, резюме

Методы *IUnknown* дают нам полный контроль над интерфейсами. Как мы видели в предыдущей главе, указатели на интерфейсы, поддерживаемые компонентом, можно получить через *QueryInterface*. В этой главе мы видели, как *AddRef* и *Release* управляют

временами жизни полученных интерфейсов. *AddRef* сообщает компоненту, что мы собираемся использовать интерфейс. *Release* сообщает, что использование интерфейса закончено. *Release* также предоставляет компоненту некоторую способность управлять своим временем жизни. Клиент никогда не выгружает компонент напрямую; вместо этого *Release* сигнализирует, что клиент завершил работу с интерфейсом. Если ни один из интерфейсов никем не используется, компонент может удалить себя сам.

Хотя теперь мы имеем все возможности управления интерфейсами, у нас нет одной важной составляющей компонентнойархитектуры — динамической компоновки. Компонент без динамической компоновки — это все равно что пожарный без каски и плаща. В следующей главе мы добавим к нашим компонентам динамическую компоновку.

6. Динамическая компоновка

Что же это получается? Еще в первой главе я говорил, как важна динамическая компоновка для построения системы из «кирпичиков». И вот мы добрались уже до шестой главы — и не только по-прежнему компонуем клиента с компонентом статически, но и располагаем их все время в одном и том же файле! На самом деле у меня были основательные причины отложить обсуждение динамической компоновки. Главная из них в том, что пока мы не реализовали полностью *IUnknown*, клиент был слишком сильно связан с компонентом.

Сначала компонент нельзя было изменить так, чтобы не потребовалось изменять и клиент. Затем при помощи *QueryInterface* мы перешли на следующий уровень абстракции и представили компонент как набор независимых интерфейсов. Раздробив компонент на интерфейсы, мы сделали первый шаг к тому, чтобы раздробить монолитное приложение. Затем нам понадобился способ управления временем жизни компонента. Подсчитывая ссылки на каждый интерфейс, клиент управляет их временем жизни, компонент же сам определяет, когда ему себя выгрузить. Теперь, когда мы реализовали *IUnknown*, клиент и компонент связаны не титановой цепью, а тонкой ниткой. Столь непрочная связь уже не мешает компоненту и клиенту изменяться, не задевая друг друга.

В этой главе мы попробуем поместить компонент в DLL. Обратите внимание — я не сказал, что мы собираемся *сделать* компонент DLL. Компонент — это не DLL, думать так значило бы слишком ограничивать концепцию компонента. DLL для компонента — сервер, или средство доставки. Компонент — это набор интерфейсов, которые реализованы в DLL. DLL — это грузовик, а компонент — груз.

Чтобы познакомиться с динамической компоновкой, мы посмотрим, как клиент создает компонент, содержащийся в DLL. Затем мы возьмем листинг 5.1 из гл. 5 и разобьем его на отдельные файлы для клиента и
компонента. Разобрав полученный код, мы создадим три разных клиента и три разных компонента, использующие разные комбинации трех интерфейсов. Для чего все это? В качестве грандиозного финала мы сконструируем компанию клиентов и компонентов, где каждый сможет общаться с каждым.

Если Вы уже знакомы с DLL, то большая часть содержания этой главы Вам известна. Однако Вы можете полюбопытствовать, как я «растащу» клиент и компонент по разным файлам (раздел «Разбиваем монолит»). Надеюсь, Вам понравится сочетание разных клиентов и компонентов в разделе «Связки объектов» в конце главы.

Создание компонента

В этом разделе мы увидим, как компонент динамически компонуется клиентом. Мы начнем с клиента, создающего компонент. Это временная мера; в последующих главах мы увидим, как изолировать клиент от компонента еще сильнее.

Прежде чем запросить указатель на интерфейс, клиент должен загрузить DLL в свой процесс и создать компонент. В гл. 4 функция *CreateInstance* создавала компонент и возвращала клиенту указатель на интерфейс *IUnknown*. Это единственная функция в DLL, с которой клиент должен быть скомпонован явно. Ко всем прочим функциям компонента клиент может получить доступ через указатель на интерфейс. Таким образом, чтобы
клиент мог вызывать функцию *CreateInstance*, ее надо экспортировать.

Экспорт функции из DLL

Экспорт функции из DLL осуществляется без проблем. Сначала необходимо обеспечить использование компоновки C (C linkage), пометив функцию как *extern “C”*. Например, функция *CreateInstance* в файле CMPNT1.CPP выгляди так:

```
//
// Функция создания
//
extern "C" IUnknown* CreateInstance()
{
  IUnknown* pI = (IUnknown*)(void*)new CA;
  PI->AddRef();
  return pI;
}
```

Слово *extern "C"* в описании нужно, чтобы компилятор C++ не «довешивал» к имени функции информацию о типе. Без *extern "C"* Microsoft Visual C++ 5.0 превратит *CreateInstance* в

```
?CreateInstance@@YAPAUIUnknown@@XZ
```

Другие компиляторы используют иные схемы дополнения имени информацией о типе. На дополненные имена нет стандарта, так что они не переносимы. Кроме того, работать с ними — изрядная морока.

Дамп экспортов

Если Вы пользуетесь Microsoft Visual C++, то при помощи DUMPBIN.EXE можете получить листинг символов, экспортированных из DLL. Следующая команда

```
dumpbin –exports Cmpnt1.dll
```

генерирует для CMPNT1.DLL такие результаты:

```
Microsoft (R) COFF Binary File Dumper Version 4.20.6281
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.

Dump of file Cmpnt1.dll

File Type: DLL

Section contains the following Exports for Cmpnt1.dll

      0 characteristics
      325556C5 time date stamp Fri Oct 04 11:26:13 1996
      0.00 version
      1 ordinal base
      1 number of functions
      1 number of names

      ordinal hint name

      1 0 CreateInstance (00001028)

Summary

      7000 .data
      1000 .idata
      3000 .rdata
      2000 .reloc
      10000 .text
```

Конечно, чтобы экспортировать функцию, недостаточно пометить ее как *extern "C"*. Необходимо еще сообщить компоновщику, что функция экспортируется. Для этого

надо создать надоедливый файл DEF. Файлы DEF так надоедливы потому, что очень легко позабыть внести в файл имя функции; если же Вы об этом забыли, компоновка этой функции будет невозможна. Из-за такой забывчивости я лишился изрядной части волос.

Создавать файлы DEF очень легко. Вы можете скопировать их из примеров и изменить несколько строк. DEF файл для CMPNT1.DLL показан в листинге 6.1.

CMPNT1.DEF

```
;
; Файл определения модуля для Cmpnt1
;

LIBRARY Cmpnt1.dll
DESCRIPTION '(c)1996-1997 Dale E. Rogerson'

EXPORTS
      CreateInstance @1 PRIVATE
```

Листинг 6.1 *В файле определения модуля перечислены функции, экспортированные динамически компонуемой библиотекой*

Все, что Вам нужно сделать, — перечислить экспортируемые функции в разделе *EXPORTS* данного файла. При желании можно назначить каждой функции порядковый номер (ordinal number). В строке *LIBRARY* следует указать фактическое имя DLL.

Таковы основы экспорта функций из DLL. Теперь мы посмотрим, как загрузить DLL и обратиться к функции.

Загрузка DLL

Файлы CREATE.H и CREATE.CPP реализуют функцию *CreateInstance*. *CreateInstance* принимает имя DLL в качестве параметра, загружает эту DLL и пытается вызвать экспортированную функцию с именем *CreateInstance*.

Соответствующий код показан в листинге 6.2.

CREATE.CPP

```
//
// Create.cpp
//

#include <iostream.h>
#include <unknwn.h> // Объявление IUnknown
#include "Create.h"

typedef IUnknown* (*CREATEFUNCPTR)();

IUnknown* CallCreateInstance(char* name)
{
  // Загрузить в процесс динамическую библиотеку
  HINSTANCE hComponent = ::LoadLibrary(name);
  if (hComponent == NULL)
  {
    cout << "CallCreateInstance:\tОшибка: Не могу загрузить компонент"
         << endl;
    return NULL;
  }
```

```
  // Получить адрес функции CreateInstance
  CREATEFUNCPTR CreateInstance
       = (CREATEFUNCPTR)::GetProcAddress(hComponent, "CreateInstance");
  if (CreateInstance == NULL)
  {
    cout << "CallCreateInstance:\tОшибка: "
         << "Не могу найти функцию CreateInstance"
         << endl;
    return NULL;
  }
  return CreateInstance();
}
```

Листинг 6.2 *Используя* LoadLibrary *и* GetProcAddress*, клиент может динамически компоноваться с компонентом*

Для загрузки DLL *CreateInstance* вызывает функцию Win32 *LoadLibrary*:

```
HINSTANCE LoadLibrary(
  LPCTSTR lpLibFileName // Имя файла DLL
);
```

LoadLibrary принимает в качестве параметра имя файла DLL и возвращает описатель загруженной DLL. Функция Win32 *GetProcAddress* принимает этот описатель и имя функции (*CreateInstance*), возвращая адрес последней:

```
FARPROC GetProcAddress(
  HMODULE hModule, // Описатель модуля DLL
  LPCSTR lpProcName // Имя функции
}
```

С помощью этих двух функций клиент может загрузить DLL в свое адресное пространство и получить адрес *CreateInstance*. Имея этот адрес, создать компонент и получить указатель на его *IUnknown* не составляет труда. *CallCreateInstance* приводит возвращенный указатель к типу, пригодному для использования, и, в соответствии со своим назначением, вызывает *CreateInstance*. Но *CallCreateInstance* слишком тесно привязывает клиент к реализации компонента. От клиента нельзятребовать знания имени DLL, в которой реализован компонент. Нам нужна и возможность перемещать компонент из одной DLL в другую или даже из одного каталога в другой.

Почему можно использовать DLL

Почему DLL можно использовать для размещения компонентов? Потому, что DLL используют адресное пространство приложения, с которым скомпонованы.

Как уже обсуждалось выше, клиент и компонент взаимодействуют через интерфейсы. Интерфейс — это по сути таблица указателей на функции. Компонент выделяет память для vtbl и инициализирует ее адресами всех функций. Чтобы использовать vtbl, клиент должен иметь доступ к выделенной для нее компонентом памяти. Клиент также должен «понимать» адреса, помещенные компонентом в vtbl. В Windows клиент может работать с vtbl, так как динамически компонуемая библиотека использует то же адресное пространство, что и он сам.

В Windows исполняющаяся программа называется *процессом*. Каждое приложение (EXE) исполняется в отдельном процесса, и у каждого процесса имеется свое адресное пространство в 4 Гбайт. Адрес в одном процессе отличен от того же адреса в другом процессе. Указатели не могут передаваться из одного приложения в другое, поскольку они

находятся в разных адресных пространствах. Пусть у нас есть некий адрес, скажем, дом 369 по Персиковой аллее. Этот дом может оказаться как супермаркетом в Атланте, так и кофейней в Сиэтле. Если не указан город, адрес на самом деле не имеет смысла. В рамках этой аналогии процессы — это города. Указатели в двух процессах могут иметь одно и то же значение, но фактически они будут указывать на разные участки памяти.

К счастью, динамически компонуемая библиотека располагается в том же процессе, что и использующее ее приложение. Поскольку и DLL, и EXE используют один и тот же процесс, они используют одно и то же адресное пространство. По этой причине DLL часто называют *серверами внутри процесса (in-proc server)*. В гл. 11 мы рассмотрим сервера *вне процесса (out-of-proc)*, или *локальные и удаленные* серверы, которые реализуются как EXE-модули. Серверы вне процесса имеют адресные пространства, отличные от адресных пространств своих клиентов, но мы по-прежнему будем использовать DLL для поддержки связи такого сервера с его клиентом. На рис. 6.1 показано размещение DLL в адресном пространстве ее клиентского приложения.

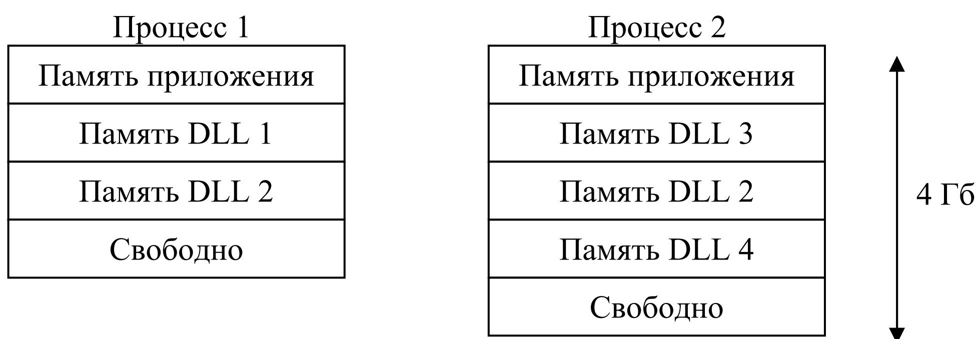


Рис. 6.1 *Динамически компонуемые библиотеки резмещаются в адресном пространстве процесса, содержащего приложение, с которым они скомпонованы*

Важно отметить, что после того, как клиент получил у компонента указатель на интерфейс, все, что их связывает, — это двоичная «развертка» интерфейса. Когда клиент запрашивает у компонента интерфейс, он запрашивает участок памяти определенного формата. Возвращая указатель на интерфейс, компонент сообщает клиенту, где находится этот участок. Поскольку интерфейс располагается в памяти, доступной и клиенту, и компоненту, ситуация аналогична той, когда клиент и компонент расположены в одном и том же EXE файле. С точки зрения клиента, единственное различие динамической и статической компоновки состоит в способе, которым он получает указатель на интерфейс.

В гл. 6 и 7 мы разъединим клиент и компонент, используя более общий и гибкий метод создания компонентов. В гл. 7 функция *CoCreateInstance* библиотеки COM заменит *CallCreateInstance*, пока же *CallCreateInstance* нам будет достаточно.

Разбиваем монолит

В этом разделе мы выясним, как можно разделить программу листинга 5.1 на несколько файлов, которые мы затем рассмотрим по отдельности. На рис. 6.2 показаны файлы, содержащие по одному клиенту и компоненту.

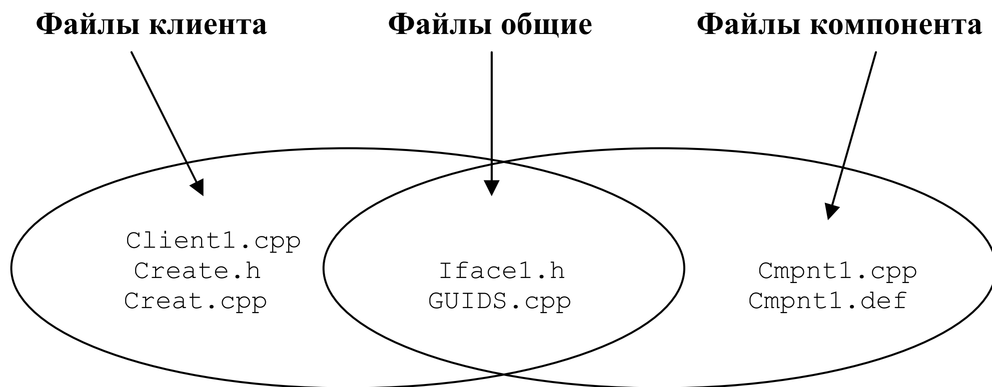


Рис. 6.2 *Файлы клиента и компонента*

Теперь клиент находится в файле CLIENT1.CPP. Он включает файл CREATE.H и компонуется вместе с файлом CLIENT1.CPP. Эти два файла инкапсулируют создание компонента, находящегося в DLL. (Файл CREATE.CPP мы уже видели в листинге 6.2.) В гл. 8 два этих файла исчезнут, их заменят функции, предоставляемые библиотекой COM.

Компонент теперь размещается в файле CMPNT1.CPP. Для динамической компоновки требуется файл определения модуля, в котором перечисляются функции, экспортируемые из DLL. Это файл CMPNT1.DEF, приведенный в листинге 6.1.

Компонент и клиент используют два общих файла. Файл IFACE.H содержит объявления всех интерфейсов, поддерживаемых CMPNT1. Там же содержатся объявления для идентификаторов этих интерфейсов. Определения данных идентификаторов находятся в файле GUIDS.CPP (потерпите, о GUID мы поговорим в следующей главе).

Собрать клиент и компонент можно с помощью следующих команд:

```
cl Client.cpp Create.cpp GUIDS.cpp UUID.lib
cl /LD Cmpnt1.cpp GUIDS.cpp UUID.lib Cmpnt1.def
```

Тексты программ

Теперь давайте рассмотрим код, особенно клиента, поскольку по-настоящему новое и интересное находится именно там. Код, реализующий клиент, представлен в листинге 6.3. Клиент запрашивает у пользователя имя файла используемой DLL. Это имя он передает функции *CallCreateInstance*, которая загружает DLL и вызывает экспортированную из нее функцию *CreateInstance*.

CLIENT1.CPP

```
//
// Client1.cpp
// Комиляция: cl Client1.cpp Create.cpp GUIDs.cpp UUID.lib
//

#include <iostream.h>
#include <objbase.h>
#include "Iface.h"
#include "Create.h"

void trace(const char* msg) { cout << "Клиент 1:\t" << msg << endl; }
```

```
//
// Клиент1
//
int main()
{
  HRESULT hr;

  // Считать имя компонента
  char name[40];
  cout << "Введите имя файла компонента [Cmpnt?.dll]: ";
  cin >> name;
  cout << endl;

  // Создать компонент вызовом функции CreateInstance из DLL
  trace("Получить указатель на IUnknown");
  IUnknown* pIUnknown = CallCreateInstance(name);
  if (pIUnknown == NULL)
  {
    trace("Вызов CallCreateInstance неудачен");
    return 1;
  }

  trace("Получить интерфейс IX");
  IX* pIX;
  hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    trace("IX получен успешно");
    pIX->Fx(); // Использовать интерфейс IX
    pIX->Release();
  }
  else
  {
    trace("Не могу получить интерфейс IX");
  }

  trace("Освободить интерфейс IUnknown");
  pIUnknown->Release();
  return 0;
}
```

Листинг 6.3 *Клиент* запрашивает имя *DLL, содержащей компонент. Он загружает DLL, создает компонент и работает с его интерфейсами.*

В листинге 6.4 приведен код компонента. За исключением спецификации *extern "C"* для *CreateInstance*, он остался практически неизменным. Только теперь компонент находится в своем собственном файле — CMPNT1.CPP. CMPNT1.CPP компилируется с использование флажка /LD. Кроме того, он компонуется с CMPNT1.DEF, который мы уже видели в листинге 6.1.

CMPNT1.CPP

```
//
// Cmpnt1.cpp
// Компиляция: cl /LD Cmpnt1.cpp GUIDs.cpp UUID.lib Cmpnt1.def
//

#include <iostream.h>
#include <objbase.h>
#include "Iface.h"
```

```
void trace(const char* msg) { cout << "Компонент 1:\t" << msg << endl; }
//
// Компонент
//
class CA : public IX
{
  // Реализация IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
  // Реализация интерфейса IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }

public:
  // Конструктор
  CA() : m_cRef(0) {}
  // Деструктор
  ~CA() { trace("Ликвидировать себя"); }

private:
  long m_cRef;
};


HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    trace("Возвратить указатель на IUnknown");
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    trace("Возвратить указатель на IX");
    *ppv = static_cast<IX*>(this);
  }
  else
  {
    trace("Интерфейс не поддерживается");
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}


ULONG __stdcall CA::AddRef()
{
  return InterlockedIncrement(&m_cRef);
}


ULONG __stdcall CA::Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}
```

```
//
// Функция создания
//
extern "C" IUnknown* CreateInstance()
{
  IUnknown* pI = static_cast<IX*>(new CA);
  pI->AddRef();
  return pI;
}
```

Листинг 6.4 *Компонент, расположенный теперь в своем файле, практически не изменился по сравнению с гл. 5.*

Теперь нам осталось взглянуть лишь на два общих файла — IFACE.H и GUIDS.CPP. В файле IFACE.H объявлены все интерфейсы, используемые клиентом и компонентом.

IFACE.H

```
//
// Iface.h
//

// Интерфейсы
interface IX : IUnknown
{
  virtual void __stdcall Fx() = 0;
};

interface IY : IUnknown
{
  virtual void __stdcall Fy() = 0;
};

interface IZ : IUnknown
{
  virtual void __stdcall Fz() = 0;
};

// Предварительные объявления GUIDs
extern "C"
{
  extern const IID IID_IX;
  extern const IID IID_IY;
  extern const IID IID_IZ;
}
```

Листинг 6.5 *Объявления интерфейсов*

Как видите, клиент и компонент по-прежнему используют интерфейсы *IX*, *IY* и *IZ*. Идентификаторы этих интерфейсов объявлены в конце IFACE.H. IID будут обсуждаться в следующей главе. Определения идентификаторов интерфейсов находятся в файле GUIDS.CPP, который показан в листинге 6.6.

GUIDS.CPP

```
//
// GUIDs.cpp - Идентификаторы интерфейсов
//
#include <objbase.h>
extern "C"
{
  // {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
  extern const IID IID_IX =
         {0x32bb8320, 0xb41b, 0x11cf,
         {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
  // {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
  extern const IID IID_IY =
         {0x32bb8321, 0xb41b, 0x11cf,
         {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
  // {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
  extern const IID IID_IZ =
         {0x32bb8322, 0xb41b, 0x11cf,
         {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
  // extern необходим, чтобы для констант C++ была выделена память
}
```

Листинг 5-6 *Идентификаторы интерфейсов определены в* GUIDS.CPP. *С этим файлом компонуются и клиент, и компонент.*

Это были детали реализации компонента в DLL. Давайте немного поиграем с такими компонентами.

Связки объектов

Теперь Вы можете поиграть с компонентами и посмотреть, как они динамически компонуются. В табл. 6.1 показан набор интерфейсов, поддерживаемых каждым клиентом и компонентом.

Таблица 6.1 *Эта таблица показывает, какие интерфейсы поддерживаются каждым клиентом и компонентом*

	IX	*IY*	*IZ*	
Клиент 1	*x*			Компонент 1
Клиент 2	*x*	*x*		Компонент 2
Клиент 3	*x*	*x*	*x*	Компонент 3

Каждый из клиентов при запуске спрашивает, какой компонент он должен использовать. Введите имя компонента и нажмите <Enter>. Клиент с соответствующим партнером. Затем он запросит у компонента каждый известный ему интерфейс. Если компонент поддерживает интерфейс, клиент вызовет функцию этого интерфейса. Ниже приведен пример работы Клиента 2 и Компонентом 2 и Клиента 3 с Компонентом 1.

c:\client2

Введите имя файла компонента [Cmpnt?.dll]: cmpnt2.dll
Клиент 2: Получить указатель на IUnknown
Компонент 2: Возвратить указатель на IUnknown
Клиент 2: Получить интерфейс IX

Компонент 2: Возвратить указатель на IX
Клиент 2: IX получен успешно

Fx
Клиент 2: Получить интерфейс IY
Компонент 2: Возвратить указатель на IY
Клиент 2: IY получен успешно

Fy
Клиент 2: Освободить интерфейс IUnknown
Компонент 2: Ликвидировать себя

C:\client3

Введите имя файла компонента [Cmpnt?.dll]: cmpnt1.dll
Клиент 3: Получить указатель на IUnknown
Клиент 3: Получить интерфейс IX
Компонент 1: Возвратить указатель на IX
Клиент 3: IX получен успешно

Fx
Клиент 3: Получить интерфейс IY
Компонент 1: Интерфейс не поддерживается
Клиент 3: Не могу получить интерфейс IY
Клиент 3: Получить интерфейс IZ
Компонент 1: Интерфейс не поддерживается
Клиент 3: Не могу получить интерфейс IZ
Клиент 3: Освободить интерфейс IUnknown
Компонент 1: Ликвидировать себя

Компонент 2 реализует все интерфейсы, нужные клиенту 2. Компонент 1 реализует только *IX*, тогда как Компоненту 3 нужны все три интерфейса: *IX*, *IY* и *IZ*. Попробуйте другие комбинации компонентов и клиентов. Мы успешно создали архитектуру, которая позволяет подключать друг к другу компоненты и клиенты во время выполнения.

Негибкое связывание, резюме

В этой главе мы добавили нашим компонентам одно свойство — динамическую компоновку. Поместив компоненты в DLL, мы можем заменять их во время выполнения. Как Вы видели из примеров, один клиент может легко работать с разными компонентами без перекомпоновки или перекомпиляции. Динамическая компоновка в сочетании с хорошо спроектированными интерфейсами может обеспечить создание невероятно гибких приложений, которые будут эволюционировать с течением времени. Как бы ни были гибки наши компоненты, по-прежнему остается один момент, в котором гибкости не хватает, — момент создания. *CallCreateInstance* требует, чтобы клиент знал имя DLL, в которой реализован компонент. Имя DLL — это деталь реализации, которую нам хотелось бы скрыть от клиента. Компонент должен быть способен изменить имя DLL, в которую он погружен, и не повлиять на клиентов. Хотелось бы также поддерживать в одной DLL несколько компонентов. Эти вопросам и посвящены следующие две главы.

7. HRESULT, GUID, Реестр и другие детали

Дух братьев Райт все еще жив. Каждый год сотни людей в своих гаражах строят самолеты из наборов «Сделай сам». Они делают не пластиковые игрушки, радиоуправляемые модели или легковесные матерчатые конструкции. Они строят современные двухместные самолеты с полностью закрытой кабиной из современнейших композитных материалов. По правилам FAA (Федерального авиационного агенства) достаточно, чтобы производитель набора выполнил только 49% всей работы по постройке самолета. Оставшийся 51% конструктор-любитель делает сам.

Постройка 51% самолета занимает, в зависимости от модели, примерно от 250 до 5000 часов. Большинство производителей предлагают наборы «для быстрого приготовления», в которых многие части уже собраны, например, сварены рамы и пропитаны детали из композитных материалов. Используя такие заготовки, можно быстро сделать нечто, похожее на самолет. Однако это будет еще далеко не настоящий самолет. Куча времени уходит на разные мелочи — установку панели управления и приборов, сидений, ремней безопасности, огнетушителей, покрытия, сигнализации, табличек, кабелей управления, электропроводки, лампочек, батарей, брандмауэров, вентиляционных люков, крыши пилотской кабины, отопителя, окон, замков и ручек на дверях и еще многого другого.

Многие энтузиасты самолетостроения приходят в уныние, разочаровываются и даже бросают это занятие, потратив на сборку многие часы. Точно так же многие начинают изучать сложный предмет, например COM, лишь затем, чтобы захлебнуться в деталях и все бросить. В первых пяти главах книги я пытался максимально избегать подробностей, чтобы Вы сосредоточились на общей картине. В этой главе я собираюсь обсудить некоторые их тех деталей, которые раньше пропускал или скрывал. Я хочу рассмотреть и другие детали, которые понадобятся нам в следующих главах.

Сначала мы обсудим HRESULT — тему, впервые возникшую в гл. 3 в связи с *QueryInterface*. Затем мы рассмотрим GUID. Один из примеров GUID — структура IID, передаваемая *QueryInterface*. После обсуждения этих типов мы познакомимся с тем, как компоненты публикуют в Реестре данные о своем местонахождении (это позволяет клиентам находить и создавать компоненты). В заключение мы рассмотрим некоторые полезные функции и утилиты библиотеки COM.

HRESULT

Во всех самолетах есть приборы, и самодельные самолеты — не исключение. Хотя в некоторых таких самолетах роль приборов играет компьютер с цветным графическим дисплеем (иногда даже «под» Windows NT), обычно ставят что-нибудь подешевле. Металлическая полоса, например, — простейший индикатор скорости. Чем быстрее Вы летите, тем сильнее она изгибается.

Хотя приборы и могут сообщить в деталях, что происходит с самолетом и отдельными системами, их основное назначение — предупреждать пилота об опасности. На индикаторе скорости есть красная полоска, отмечающая слишком высокую (или низкую) скорость. Часто приборы снабжаются аварийными лампочками и зуммерами. У компонентом COM нет приборов. Вместо шкал или лампочек для сообщений о текущем состоянии дел они используют HRESULT. *QueryInterface* возвращает HRESULT. И, как мы увидим в оставшейся части книги, большинство функций интерфейсов COM также возвращает HRESULT. Хотя из названия *HRESULT* можно было бы заключить, что это описатель (handle) результата, на самом деле это не так. HRESULT — это 32-разрядное значение, разделенное на три поля. Значение полей, составляющих HRESULT, поясняет рис. 7.1. Название возникло по историческим причинам; просто расшифровывайте его как «вот результат» (here’s the result), а не «описатель результата» (handle of result).

Определенные системой значения HRESULT содержатся в заголовочном файле Win32 WINERROR.H. В начале файла расположены коды ошибок Win32, так что его нужно пролистать, чтобы добраться до HRESULT. HRESULT похож на код ошибки Win32, но это не одно и то же, и смешивать их не следует.

Старший бит HRESULT, как показано на рис. 6-1, отмечает, успешно или нет выполнена функция. Это позволяет определить много кодов возврата и для успеха, и для неудачи. Последние 16 битов содержат собственно код возврата. Остальные 15 битов содержат дополнительную информацию о типе и источнике ошибки.

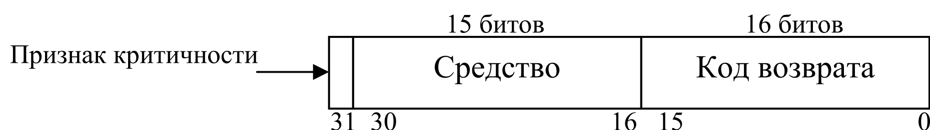


Рис. 7.1 *Формат HRESULT*

В табл. 7.1 приведены наиболее часто используемые коды. По соглашению в названиях успешных кодов содержится *S_*, а в названиях кодов ошибок — *E_*.

Таблица 7.1 *Распространенные значения HRESULT*

Название	**Значение**
S_OK	Функция отработала успешно. В некоторых случаях этот код также означает, что функция возвращает логическую истину. Значение S_OK равно 0
NOERROR	То же, что S_OK
S_FALSE	Функция отработала успешно и возвращает логическую ложь. Значение S_FALSE равно 1
E_UNEXPECTED	Неожиданная ошибка
E_NOIMPL	Метод не реализован
E_NOINTERFACE	Компонент не поддерживает запрашиваемый интерфейс. Возвращается *QueryInterface*
E_OUTOFMEMORY	Компонент не может выделить требуемый объем памяти
E_FAIL	Ошибка по неуказанной причине

Обратите внимание, что значение S_FALSE равно 1, а значение S_OK — 0. Это противоречит обычной практике программирования на C/C++, где 0 — это ложь, а не-0 — истина. Поэтому при использовании HRESULT обязательно явно сравнивайте коды возврата с S_FALSE или S_OK.

Пятнадцать битов — с 30-го по 16-й — содержат идентификатор средства (facility). Он указывает, какая часть операционной системы выдает данный код возврата. Поскольку операционную систему разрабатывает Microsoft, она зарезервировала право определения идентификаторов средств за собой. Идентификаторы средств, определенные в настоящее время, приведены в табл. 7.2.

Таблица 7.2 *Идентификаторы средств, определенные в настоящее время*

FACILITY_WINDOWS	8
FACILITY_STORAGE	3
FACILITY_SSPI	9
FACILITY_RPC	1
FACILITY_Win32	7
FACILITY_CONTROL	10
FACILITY_NULL	0
FACILITY_ITF	4
FACILITY_DISPATCH	2
FACILITY_CERT	11

Идентификатор средства освобождает, например, разработчиков Microsoft, занятых RPC (FACILITY_RPC), от необходимости согласовывать значения кодов возврата с теми, кто работает над управляющими элементами ActiveX (FACILITY_CONTROL). Поскольку группы разработчиков используют разные идентификаторы средств, коды возврата разных средств не будут конфликтовать. Разработчикам специализированных интерфейсов повезло меньше.

Все идентификаторы средств, кроме FACILITY_ITF, задают определенные COM универсальные коды возврата. Эти коды всегда и везде одни и те же. FACILITY_ITF — исключение; ему отвечают коды, специфичные для данного интерфейса. Чтобы определить средство для данного HRESULT, используйте макрос HRESULT_FACILITY, определенный в WINERROR.H. Как Вы увидите в разделе «Определение собственных кодов возврата», коды FACILITY_ITF не уникальны и могут иметь разные значения в зависимости от интерфейса, возвратившего код. Но прежде чем определять собственные коды, давайте рассмотрим использование HRESULT.

Поиск HRESULT

Как уже отмечалось, определение всех кодов состояния COM (и OLE — точнее, уже ActiveX), генерируемых системой в настоящее время, содержится в WINERROR.H. Обычно коды заданы как шестнадцатеричные числа; запись для E_NOINTERFACE выглядит так:

```
// MessageID: E_NOINTERFACE
//
// MessageText:
//
// Данный интерфейс не поддерживается*
//
#define E_NOINTERFACE 0x80004002L
```

Однако если идентификатор средства HRESULT равен FACILITY_WIN32, Вы можете не найти его среди других. Часто это будет код ошибки Win32, преобразованный в HRESULT. Чтобы найти его значение, отыщите код ошибки Win32, совпадающий с последними 16 битами. Пусть, например, интерфейс возвращает код ошибки 0x80070103. Число 7 в середине — это идентификатор средства FACILITY_WIN32. В файле WINERROR.H Вы не найдете этот код там, где перечислены другие HRESULT. Поэтому переведите последние 16 битов из шестнадцатеричного представления в двоичное; получится число 259, которое уже можно найти в списке кодов Win32.

```
// MessageID: ERROR_NO_MORE_ITEMS
//
// MessageText:
//
// Больше элементов нет
//
#define ERROR_NO_MORE_ITEMS 259L
```

Искать HRESULT в WINERROR.H вполне допустимо, когда мы пишем код. Однако нашим программам необходим способ получить сообщение об ошибке, соответствующее данному HRESULT, и отобразить его пользователю. Для отображения сообщений о стандартных ошибках COM (а также ActiveX, ранее OLE, и Win32) можно использовать API Win32 *FormatMessage*:

```
void ErrorMessage(LPCTSTR str, HRESULT hr)
{
  void* pMsgBuf;
  ::FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
      NULL,
      hr,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR)&pMsgBuf,
      0,
      NULL
  );

  // Отобразить строку
  cout << str << "\r\n";
  cout << "Error (" << hex << hr << "): " << (LPTSTR)pMsgBuf << endl;
  // Освободить буффер
  LocalFree(pMsgBuf);
}
```

Использование HRESULT

Как видите, использовать HRESULT несколько сложнее, чем обычные булевы коды возврата. Среди потенциально чреватых осложнениями особенностей можно назвать:

- множественность кодов как успешного, так и ошибочного завершения;
- тот факт, что коды ошибок могут изменяться.

Множественность кодов завершения

Как правило, в зависимости от обстоятельств функции возвращают различные коды как успешного, так и ошибочного завершения. Именно поэтому мы использовали SUCCEEDED и FAILED. Для проверки успешного завершения нельзя сравнивать HRESULT с каким-либо одним кодом, например S_OK; равно как и для проверки неудачного завершения HRESULT нельзя сравнивать с каким-то одним кодом, например E_FAIL. Иными словами, нельзя писать:

```
HRESULT hr = CreateInstance(...);
if (hr == E_FAIL) // Не делайте так!
  return;
hr = pI->QueryInterface(...);
if (hr == S_OK) // Не делайте так!
{
  pIX->Fx();
```

```
  pIX->Release();
}
pI->Release();
```

Вместо этого надо использовать макросы SUCCEEDED и FAILED.

```
HRESULT hr = CreateInstance(...);
if (FAILED(hr))
  return;
hr = pI->QueryInterface(...);
if (SUCCEEDED(hr))
{
  pIX->Fx();
  pIX->Release();
}
pI->Release();
```

Коды ошибок могут изменяться

После того, как Ваш клиент закончен, другие разработчики могут определить новые коды ошибок для HRESULT, с которыми столкнется клиент. Поскольку компоненты, используемые клиентом, могут меняться, могут изменяться и возвращаемые ими коды ошибок. Предположим, мы пишем компонент как сервер внутри процесса. Некоторое время спустя мы решили модернизировать его, сделав удаленным сервером на другой машине. Первая версия компонента не возвращала никаких кодов ошибок сети, тогда как вторая версия может это делать. Клиент не может заранее знать обо всех возможных ошибках, поэтому он должен быть готов к обработке неожиданных ошибок. Обрабатывайте все неожиданные ошибки так же, как E_UNEXPECTED.

С кодами успешного окончания этой проблемы нет. Набор этих кодов для Вашей функции должен быть статичным. Коды успешного завершения — часть интерфейса и поэтому не могут изменяться. Клиент, использующий интерфейс, должен быть способен понять, предсказать и обработать все возможные случаи успешного завершения, поскольку ему надо будет продолжать работу. Клиенту нет необходимости обрабатывать все возможные коды ошибок — он не обязан продолжать работу, если ему встретился неожиданный код.

HRESULT и сеть

Часто связь с удаленной машиной по сети неожиданно прерывается. Если клиент работает с удаленным компонентом, он должен уметь элегантно обрабатывать разрыв сетевого соединения. Это означает, что у каждого вызова функции, который может выполняться по сети, должен быть некий способ индикации разрыва связи. По этой причине все методы, которые могут выполняться на удаленной машине, должны возвращать HRESULT. Избегайте других типов возвращаемого значения, например:

```
double GetCordLength(double BladeSection);
```

Вместо этого возвращайте из функции HRESULT, а все результаты передавайте через выходные параметры:

```
HRESULT GetCordLength(/* in */ double BladeSection, /* out */ double*
pLength);
```

HRESULT передает клиенту информацию, необходимую для обнаружения сетевых ошибок. Вызовы функций в Автоматизации (ранее OLE Автоматизация) удовлетворяют этому требованию. Более подробно удаленные компоненты будут рассмотрены в гл. 11.

Определение собственных кодов ошибки

СОМ определяет универсальные коды возврата, таки как S_OK и E_UNEXPECTED. Разработчики интерфейсов ответственны за коды возврата, специфичные для их интерфейсов. HRESULT, содержащий специфичный для интерфейса код возврата, должен также содержать идентификатор средства FACILITY_ITF. Он указывает клиенту, что код специфичен для данного интерфейса.

Хотя смысл кода возврата, отмеченного с помощью FACILITY_ITF, специфичен для возвращающего его интерфейса, само по себе соответствующее число не уникально — возможны только 216 разных значений. Тысячи разработчиков пишут свои компоненты СОМ со своими кодами возврата. Все такие коды помечены с помощью FACILITY_ITF. Поэтому не просто с очень большой вероятностью, но и с гарантиейразные интерфейсы придадут разный смысл одним и тем же кодам возврата. Тридцати двух разрядов недостаточно, чтобы дать каждому разработчику ввести свой собственный идентификатор средства, а большая длина HRESULT снизила бы эффективность. В качестве кодов возврата GUID не являются разумной альтернативой длинным целым значениям, поскольку размер GUID слишком велик. Однако, поскольку FACILITY_ITF отмечает каждый код возврата как специфичный для интерфейса, постольку такой код возврата связан с идентификатором интерфейса (IID).

Для клиента, вызывающего функции интерфейса, возможность конфликта кодов возврата — не проблема. Клиент знает, к кому он обращается, и, таким образом, знает все коды успеха данного интерфейса. Клиент также известна большая часть кодов ошибок. Клиент должен рассматривать любой неизвестный ему код ошибки как E_UNEXPECTED. Однако проблемы начинаются, когда клиент интерфейса сам является компонентом, который пытается без изменения передать возвращаемый код успеха или ошибки своему клиенту. Последний не поймет код, поскольку не знает, к какому первоначальному интерфейсу тот относится.

Например, предположим, что первый клиент вызывает функцию *IX::Fx*, которая затем вызывает *IY::Fy*. Если *IY::Fy* возвращает HRESULT с FACILITY_ITF, то *IX::Fx* не может передать этот код первому клиенту. Данный клиент знает только о *IX* — и будет полагать, что HRESULT относится к *IX*, а не *IY*. Следовательно, *IX::Fx* должна транслировать возвращаемые *IY* значения HRESULT с FACILITY_ITF в такие значения, которые понятны первому клиенту. Для неизвестных ошибок у *IX* нет иного выбора, кроме как возвращать E_UNEXPECTED. Для кодов успешного завершения *IX* должен возвращать свои собственные, документированные коды возврата.

Вот некоторые основные правила определения собственных HRESULT:

- Не назначайте кодам возврата значения из диапазона 0x0000 — 0x01FF. Они зарезервированы для кодов FACILITY_ITF, определенных СОМ.
- Не возвращайте клиенту коды с признаком FACILITY_ITF без изменения.
- Используйте универсальные коды успеха и ошибки СОМ всегда, когда только возможно.
- Избегайте определения собственных HRESULT; вместо этого используйте выходные параметры Вашей функции.

Теперь, когда Вы получили некотороепредставление о HRESULT, создадим полный код при помощи макроса *MAKE_HRESULT*. По заданному признаку критичности, идентификатору средства и коду завершения *MAKE_HRESULT* создает HRESULT. Вот два примера:

```
MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, 512);
MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_ITF, 513);
```

По соглашению нестандартным кодам завершения дается в качестве префикса имя компонента или интерфейса. Например, двум приведенным выше кодам можно было бы дать имена

```
AIRPLANE_E_LANDINGWITHGEARUP
HELICOPTER_S_ROTORRPMGREEN
```

Сказанного о HRESULT более чем достаточно. Теперь пора снять завесу таинственности с GUID.

GUID

В США всем обычным летательным аппаратам FAA присваивает *N-номер (N number)*, который идентифицирует самолет, — как номерной знак идентифицирует Вашу машину. Этот номер уникален для каждого самолета и используется пилотом в переговорах с авиадиспетчерами. В этом разделе мы обсудим GUID, которые являются такими «опознавательными знаками» компонентов и интерфейсов. В гл. 4 я предложил Вам представлять себе IID как константу, идентифицирующую данный интерфейс. Однако, как Вы могли видеть из определения *IID_IX*, IID — это константа особого рода:

```
extern const IID IID_IX =
        {0x32bb8320, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

На самом деле IID представляет собой тип, определенный как структура длинной 128 битов (16 байтов) под названием *GUID*. GUID — аббревиатура *Globally Unique IDentifier (глобально уникальный идентификатор*; произносится как «гуид» — как первая часть в слове geoduck1 и последняя — в druid).

Зачем нужен GUID?

Почему мы используем GUID, а не длинное целое (long integer)? С помощью длинных целых можно однозначно задать 2^{32} интерфейсов. Я сильно сомневаюсь, что большее их число когда либо понадобится. Однако настоящая проблема не в том, сколь много интерфейсов мы сможем однозначно задать, но в том, как *гарантировать* уникальность идентификатора интерфейса. Если два идентификатора совпадают, клиент легко может получить от *QueryInterface* неверный указатель на интерфейс. Проблема усложняется тем, что компоненты создаются разработчиками по всему земному шару. Если Сара в Ако и Линн в Таксоне разрабатывают новые интерфейсы COM, то как им удостовериться, что идентификаторы интерфейсов не будут конфликтовать? Можно было бы договориться о чем-нибудь вроде N-номеров летательных аппаратов и учредить некое центральное агентство, наподобие FAA, для выделения идентификаторов. Централизованная организация подходит для относительно ограниченного числа летательных аппаратов; но я сомневаюсь, что какое-нибудь агентство смогло бы столь же успешно, как FAA, работать с тем количеством интерфейсов, которое необходимо для средней программы.

Для GUID есть более удачное решение. Уникальный GUID можно сгенерировать программно, без какой-либо координирующей организации. Microsoft Visual C++ предоставляет для генерации GUID две программы — утилиту командной строки UUIDGEN.EXE и диалоговую программу на VC++, GUIDGEN.EXE. Если я сейчас запущу UUIDGEN.EXE, то получу строку, представляющую некоторый GUID:

```
{166769E1-88E8-11CF-A6BB-0080C7B2D692}
```

При всяком новом запуске UUIDGEN получается иной GUID. Если Вы запустите UUIDGEN на своей машине, то получите GUID, отличный от моего. Если миллионы (я надеюсь) людей, читающих эту книгу, сейчас запустят UUIDGEN, они получат миллион разных GUID. Исходный текст GUIDGEN.EXE можно найти в примерах программ Microsoft Visual C++. Но я и так могу сказать Вам, как работает эта программа: она просто вызывает функцию библиотеки COM Microsoft *CoCreateGuid*, которая вызывает функцию RPC *UuidCreate*.

Теория GUID

GUID по определению уникален «в пространстве и во времени». Для обеспечения «географической» уникальности каждый GUID использует 48-битовое значение, уникальное для компьютера, на котором он генерируется. Обычно в качестве такого значения берется адрес сетевой платы. Такой подход гарантирует, что любой GUID, полученный на моем компьютере, будет отличаться от любого, сгенерированного на Вашем компьютере. Для тех компьютеров, в которых не установлен сетевой адаптер, используется другой алгоритм генерации уникальных значений. В каждом GUID 60 битов отведено для указания времени. Туда заносится число 100-наносекундных интервалов, прошедших с 00:00:00:00 15 октября 1582 года. Используемый в настоящее время алгоритм генерации GUID начнет выдавать повторяющиеся значения примерно в 3400 году. (Я подозреваю, что очень немногие из нынешних программ, за исключением некоторых на Фортране, еще будут использоваться в 3400 году; но я верю, что к этому времени уже выйдет Windows 2000.)

GUID придумали толковые ребята из Open Software Foundation (OSF); правда, они использовали термин UUID (Universally Unique IDentifiers — вселенски уникальные идентификаторы). UUID разработали для использования в среде распределенных вычислений (DCE, Distributed Computing Environment). Вызовы удаленных процедур (RPC) DCE используют UUID для идентификации вызываемого, т.е. практически затем же, зачем и мы. Дополнительно о генерации UUID или GUID можно прочитать в CAE Specification *X/Open DCE: Remote Procedure Call*.

Объявление и определение GUID

Поскольку размер GUID велик (128 битов), не хотелось бы, чтобы они повторялись в нашем коде повсюду. В гл. 6 GUID определялись в файле GUIDS.CPP примерно так:

```
extern const IID IID_IX =
        {0x32bb8320, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

Объявлены они были в файле IFACE.H так:

```
extern "C" const IID IID_IX;
```

Ввести для GUID два файла, один с определениями, а другой с объявлениями — изрядная морока. Чтобы определить и объявить GUID одним оператором, используйте макрос DEFINE_GUID, который определен в OBJBASE.H. Для использования DEFINE_GUID генерируйте GUID с помощью GUIDGEN.EXE. Эта программа генерирует GUID в различных форматах — выберите второй из них. Этот формат используется в следующем примере.

```
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
DEFINE_GUID(<<name>>,
      {0x32bb8320, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

Вставьте сгенерированный GUID в заголовочный файл. Замените <<name>> идентификатором, используемым в Вашем коде, — например, *IID_IX*:

```
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
DEFINE_GUID(IID_IX,
      {0x32bb8320, 0xb41b, 0x11cf,
      {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

В соответствии с определением в OBJBASE, DEFINE_GUID генерирует что-то вроде:

```
extern "C" const GUID IID_IX;
```

Однако, если после OBJBASE.H включить заголовочный файл INITGUID.H, макрос DEFINE_GUID будет раскрываться так:

```
extern "C" const IID IID_IX =
             {0x32bb8320, 0xb41b, 0x11cf,
             {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

Механизм работы этих заголовочных файлов представлен на рис. 7.2. Заголовок IFACE.H использует макрос DEFINE_GUID для объявления *IID_IX*. Идентификатор *IID_IX* определен в файле GUIDS.H. Он определен там потому, что заголовочный файл INITGUID.H включен после OBJBASE.H и перед IFACE.H. С другой стороны, в файле CMPNT.CPP *IID_IX* объявлен но не определен, поскольку заголовочный файл INITGUID.H здесь не включен.

Так как я старался сделать примеры в книге максимально ясными, то DEFINE_GUID я в них не использовал, но явно определял используемые GUID.

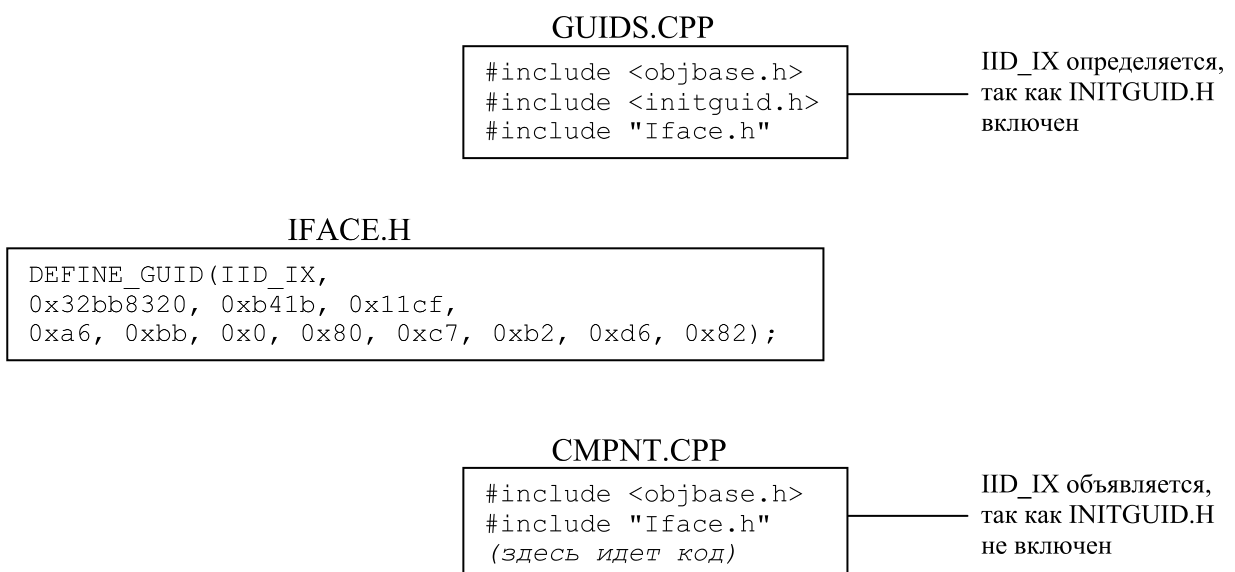


Рис. 7.2 *Включение INITGUID.H заставляет макрос DEFINE_GUID определить GUID*

Сравнение GUID

Для сравнения GUID в OBJBASE.H определен *operator==*:

```
inline BOOL operator ==(const GUID& guid1, const GUID& guid2)
{
  return !memcmp(&guid1, &guid2, sizeof(GUID));
}
```

Нам уже приходилось использовать эту операцию в *QueryInterface*. Если Вы не любите упрятывать истинный код во внешние простенькие операторы, OBJBASE.H дает определения эквивалентных по смыслу функций *IsEqualGUID, IsEqualIID* и *IsEqualCLSID*.

Использование GUID в качестве идентификаторов компонентов

Помимо уникальной идентификации интерфейсов, GUID используется и для уникальной идентификации компонентов. В гл. 6 мы определили для создания компонентов функцию *CallCreateInstance*. Параметром этой функции служит строка с именем DLL, в которой содержится компонент:

```
IUnknown* CallCreateInstance(char* name);
```

В следующей главе мы заменим эту функцию на функцию библиотеки COM *CoCreateInstance*. Последняя использует для идентификации компонента не строку, а GUID. Такой GUID в COM называется *идентификатором класса*. Чтобы отличать идентификаторы классов от IID, для них используют тип CLSID.

Передача GUID по ссылке

Поскольку размер GUID 16 байтов, мы будем передавать их не по значению, а по ссылке. Именно поэтому параметром *QueryInterface* является ссылка на константу. Если для Вас утомительно все время писать

```
const IID&
```

можете использовать эквивалентное выражение REFID. Точно так же для передачи идентификаторов классов можно использовать REFCLSID, а для передачи GUID — REFGUID.

Теперь давайте рассмотрим, как компоненты регистрируются в системе (чтобы клиенты смогли их найти и использовать).

Реестр Windows

FAA ведет реестр всех летательных аппаратов, включая самодельные. По этому реестру можно определить, кто хозяин самолета. В этой главе мы рассмотрим чем-то похожий реестр, позволяющий определить, какой DLL принадлежит данный компонент.

В гл. 6 при создании компонента мы передавали функции *CallCreateInstance* имя файла соответствующей DLL. В следующей главе мы собираемся заменить *CallCreateInstance* функцией библиотеки COM *CoCreateInstance*. Для идентификации компонента *CoCreateInstance* вместо имени файла использует CLSID (по нему определяется имя файла DLL). Компоненты помещают имена своих файлов, индексированные CLSID, в *Реестр Windows*. *CoCreateInstance* отыскивает имя файла, используя CLSID как ключ.

В реальной жизни реестр — это учетная книга для записи предметов, имен или действий. В Windows реестр — это общедоступная база данных операционной системы. Реестр содержит информацию об аппаратном и программном обеспечении, о конфигурации компьютера и о пользователях. Любая программа для Windows может добавлять и считывать информацию из Реестра; клиенты могут искать там нужные компоненты. Но прежде чем поместить свою информациюв Реестр, надо узнать, как он устроен.

Организация Реестра

Реестр имеет иерархическую структуру. Каждый ее элемент называется *разделом (key)*. Раздел может включать в себя набор подразделов, набор именованных параметров и/или один безымянный параметр — параметр *по умолчанию (default value)*. Подразделы, но не параметры, могут содержать другие подразделы и параметры. Параметры могут быть разного типа, но чаще всего мы будем записывать в Реестр строки. Структура Реестра показана на рис. 7.3.

Редактор Реестра

Реестр содержит очень много информации. По счастью, нас интересует лишь малое подмножество. Лучше всего изучать Реестр, запустив Редактор Реестра — Windows-программу, позволяющую просматривать и редактировать записи. Эта программа называется REGEDT32.EXE в Windows NT и REGEDIT.EXE в Windows 95. Одно предостережение: редактируя Реестр, *чрезвычайно легко* повредить систему, так что будьте осторожны.

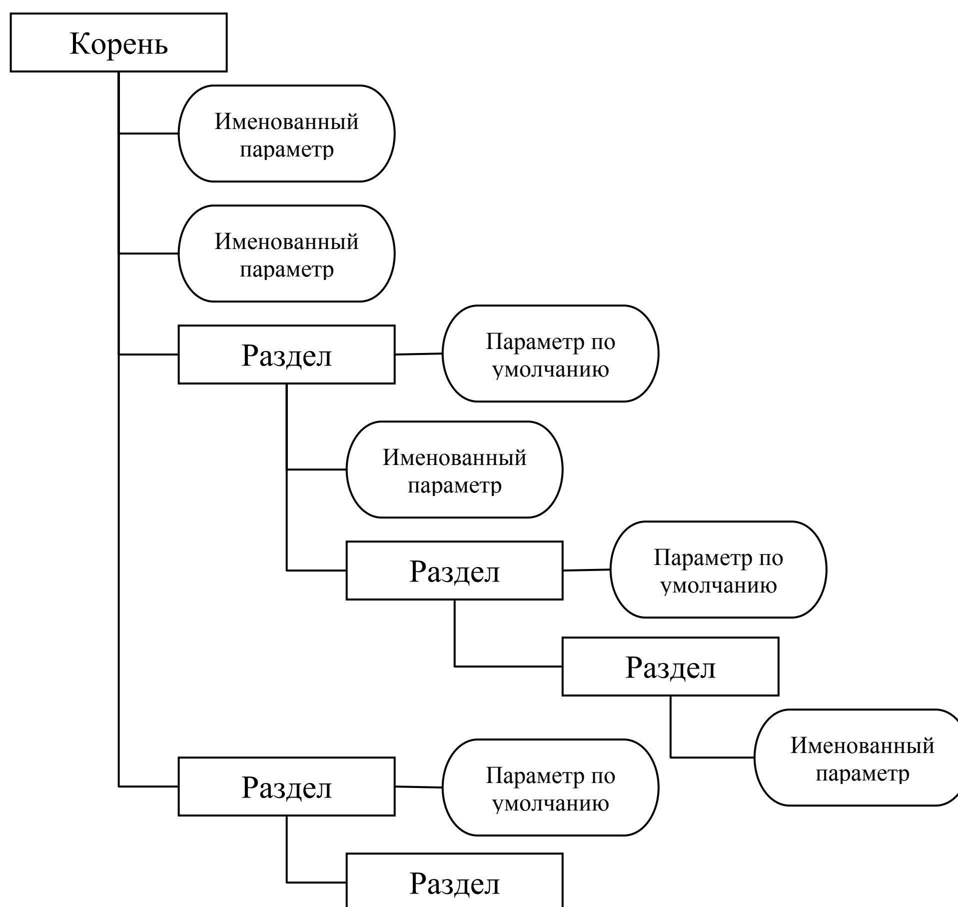


Рис. 7.3 *Структура Реестра Windows*

Необходимый минимум

COM использует только одну ветвь дерева данных Реестра: *HKEY_CLASSES_ROOT*. Ниже *HKEY_CLASSES_ROOT* отыщите раздел *CLSID*. В этом разделе перечислены CLSID всех компонентов, установленных в системе. CLSID хранится в Реестре как строка формата {xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx}. Искать CLSID в Реестре — занятие не слишком привлекательное. Поэтому в каждом разделе *CLSID* параметр по умолчанию задает «дружественное» имя компонента.

Пока в разделе каждого *CLSID* нас интересует только один подраздел — *InprocServer32*. Его параметр по умолчанию — имя файла DLL. Название *InprocServer32* используется потому, что DLL — это сервер в процессе (in-proc); она загружается в процесс клиента и предоставляет ему сервисы. На рис. 7.4 показан пример ветви CLSID Реестра.

Как видно из рисунка, в разделе Реестра *HKEY_CLASSES_ROOT\CLSID* хранится CLSID компонента Tail Rotor Simulator. Дружественное имя зарегистрировано как параметр по умолчанию для CLSID компонента. Подраздел *InprocServer32* содержит имя файла DLL — *C:\Helicopter\TailRotor.dll*.

Имя файла и CLSID — две наиболее важные составляющие данных Реестра. Для многих компонентов COM ничего больше и не потребуется. Однако в некоторых случаях нужна дополнительная информация.

Другие детали Реестра

Давайте совершим краткую экскурсию по подразделам *HKEY_CLASSES_ROOT*. Мы уже знакомы с *CLSID*, и далее мы рассмотрим, какая дополнительная информация для классов хранится в этом подразделе. В начале *HKEY_CLASSES_ROOT* Вы можете видеть группу расширений имен файлов, зарегистрированных разными программами. После расширений следует множество других имен. По большей части это так называемые *ProgID* — что расшифровывается как *программный идентификатор (program identifier)*. Мы поговорим о ProgID немного ниже. Некоторые из имен — не ProgID, а специальные разделы реестра, похожие на *CLSID*. Эти разделы связывают GUID с некоторыми другими данными, например, именами файлов. Такие разделы перечислены ниже.

- ***AppID*** — Подразделы данного раздела связывают APPID (application identifier — идентификатор приложения) с именем удаленного сервера. Этот раздел использует DCOM и будет обсуждаться в гл. 11.
- ***Component Categories*** — Эта ветвь Реестра связывает CATID (component category ID — идентификатор категории компонентов) с соответствующей категорией. Категории компонентов рассматриваются ниже.
- ***Interface*** — Данный раздел связывает IID с информацией, специфичной для интерфейса. Эта информация нужна в основном для доступа к интерфейсу «через границы» процессов. Мы рассмотрим этот раздел в гл. 11.
- ***Licenses*** — Раздел *Licenses* хранит лицензии на право использования компонентов COM. В этой книге лицензии рассматриваться не будут.
- ***TypeLib*** — Помимо других данных, библиотеки типа содержат информацию о параметрах функций- членов интерфейсов. Этот раздел связывает LIBID с именем файла, в котором хранится библиотека типа.

Библиотеки типов будут обсуждаться в гл. 12.

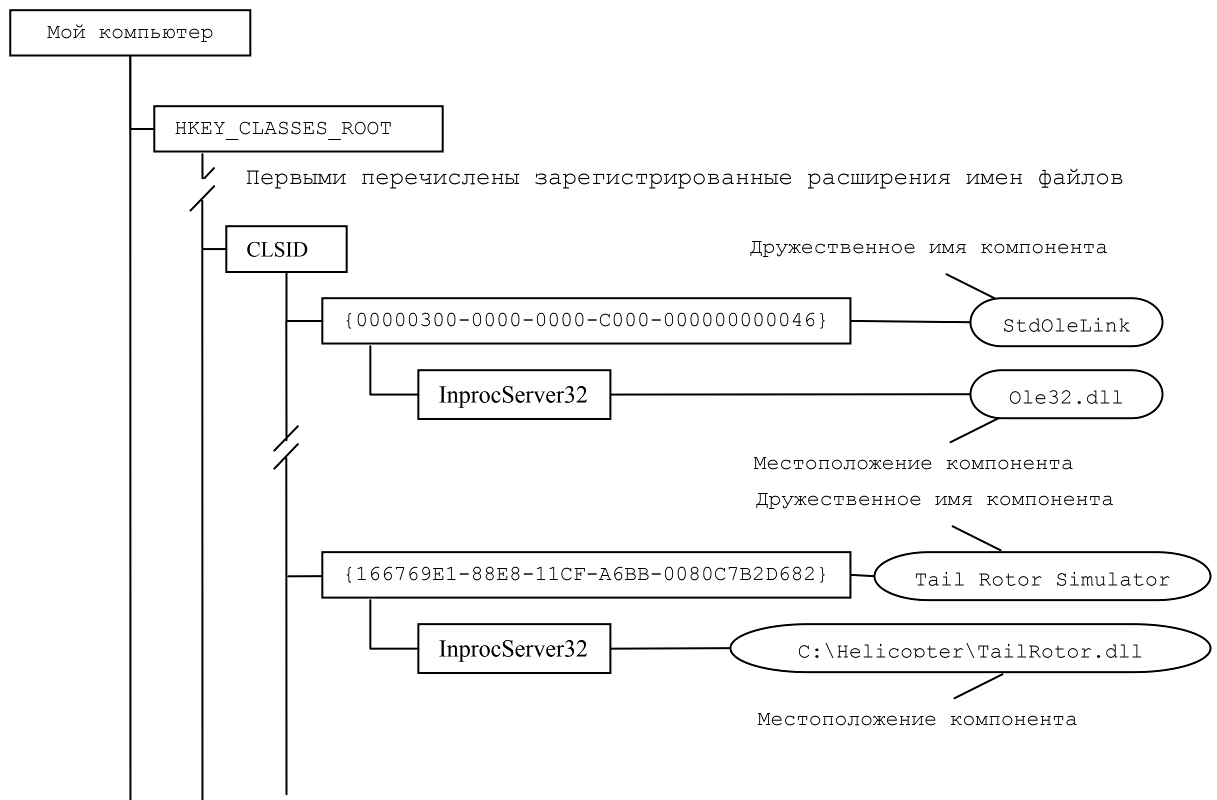


Рис. 7.4 *Структура подраздела* CLSID *Реестра*

ProgID

Теперь рассмотрим ProgID более подробно. Большая часть подразделов в ветви Реестра *HKEY_CLASSES_ROOT* — это ProgID. ProgID отображает «дружественную», понятную программисту строку в CLSID. Некоторые языки программирования, такие как Visual Basic, идентифицируют компоненты по ProgID, а не по CLSID. Уникальность ProgID не гарантируется, поэтому существует принципиальная опасность конфликта имен. Однако с ProgID легче работать. (Кроме того, некоторые языки программирования не поддерживают структур, и в них пришлось бы использовать строковое представление GUID.)

Соглашение об именовании ProgID

По соглашению ProgID имеет следующий формат:

```
<Программа>.<Компонент>.<Версия>
```

Вот несколько примеров из Реестра:

```
Visio.Application.3
Visio.Drawing.4
RealAudio.ReadAudio ActiveX Control (32-bit).1
Office.Binder.95
MSDEV.APPLICATION
JuiceComponent.RareCat.1
```

Но этот формат — лишь соглашение, а не жесткое правило, и в Реестре на моей машине полно компонентов, которые ему не следуют. Во многих случаях клиента не

интересует версия компонента, к которой он подключается. Таким образом, у компонента часто имеется ProgID, не зависящий от версии. Этот ProgID связывается с самой последней версией компонента из установленных в системе. Соглашение об именовании не зависящих от версии ProgID сводится к отбрасыванию номера версии. Пример такого ProgID, следующего соглашению, — *MSDEV.APPLICATION*.

ProgID в Реестре

ProgID и не зависящий от версии ProgID компонента приводятся в разделе CLSID. Однако основное назначение ProgID — обеспечить получение соответсвующего CLSID. Просматривать все разделы CLSID для поиска ProgID было бы неэффективно. В связи с этим ProgID указывается непосредственно и в разделе *HKEY_CLASSES_ROOT*. ProgID не предназначены для представления конечным пользователям, поэтому по умолчанию значение любого раздела *ProgID* — дружественное для пользователя имя. В разделе *ProgID* имеется подраздел с именем *CLSID*, который содержит CLSID компонента в качестве значения по умолчанию. Не зависящий от версии ProgID также приводится непосредственно в разделе *HKEY_CLASSES_ROOT*. У него есть дополнительный подраздел *CurVer*, содержащий ProgID текущей версии компонента.

На рис. 7.5 представлен расширенный пример с рис. 6-4, включающий ProgID. В раздел *CLSID* компонента добавлен раздел с именем *ProgID*, и в него помещено значение *Helicopter.TailRotor.1* — ProgID компонента. Не зависящий от версии ProgID сохранен в разделе *VersionIndependentProgID*. В данном примере не зависящий от версии ProgID — *Helicopter.TailRotor*.

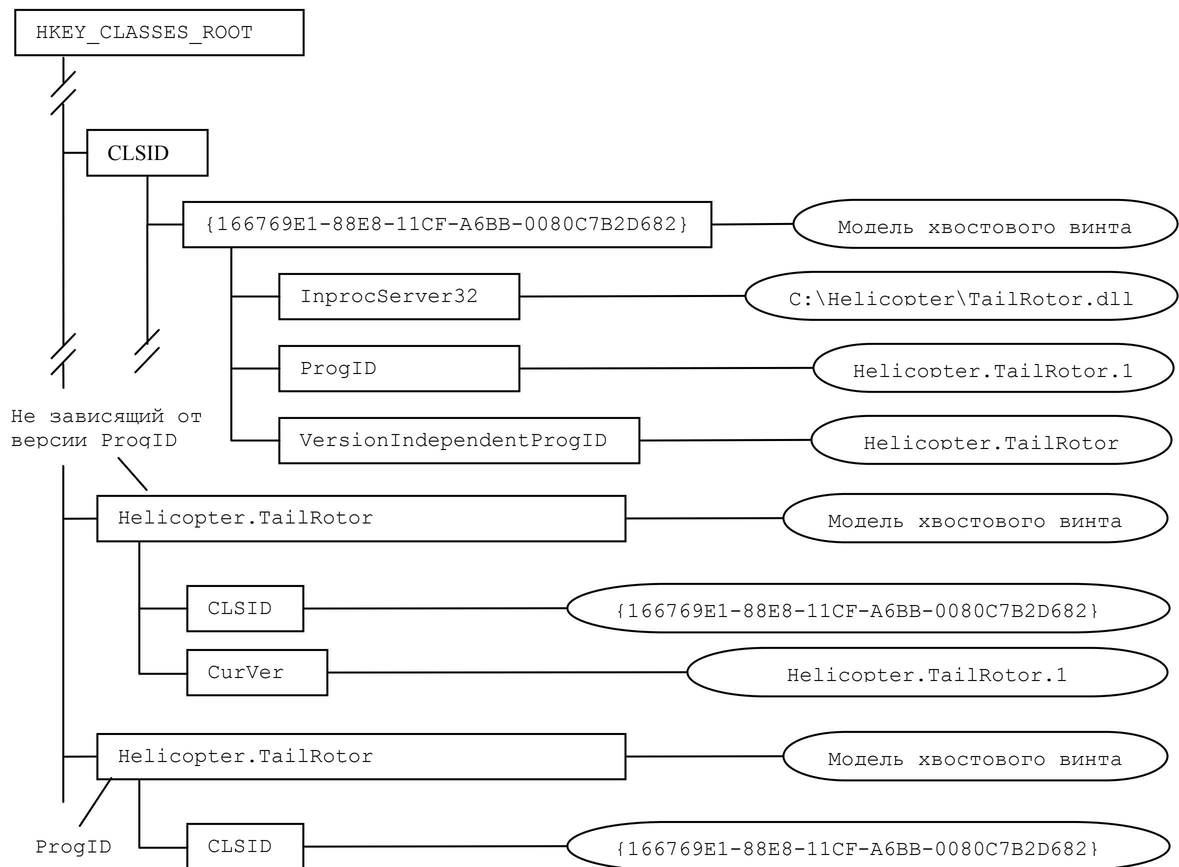


Рис. 7.5 *Организация разделов Реестра, в которых содержится информация, имеющая отношение к ProgID*

На рисунке также показаны отдельные разделы *Helicopter.TailRotor* и *Helicopter.TailRotor.1*, расположенные непосредственно в *HKEY_CLASSES_ROOT*. В разделе *Helicopter.TailRotor.1* имеется единственный подраздел — *CLSID*, который содержит CLSID компонента. Не зависящий от версии ProgID *Helicopter.TailRotor* содержит подразделы *CLSID* и *CurVer*. Значение по умолчанию подраздела *CurVer* — ProgID текущей версии компонента, *Helicopter.TailRotor.1*.

От ProgID к CLSID

После того, как Вы поместили в Реестр нужную информацию, получить CLSID по ProgID и наоборот легко. Библиотека COM предоставляет две функции — *CLSIDFromProgID* и *ProgIDFromCLSID*, — которые производят необходимые манипуляции с Реестром:

```
CLSID clsid;
CLSIDFromProgID("Helicopter.TailRotor", &clsid);
```

Саморегистрация

Каким образом информация о компоненте попадает в Реестр Windows? Так как DLL знает о содержащемся в ней компоненте, она может поместить эту информацию в Реестр. Но, поскольку DLL ничего не делает сама по себе, Вам следует экспортировать следующие две функции:

```
STDAPI DllRegisterServer();
STDAPI DllUnregisterServer();
```

STDAPI определен в OBJBASE.H как

```
#define STDAPI EXTERN_C HRESULT STDAPICALLTYPE
```

что раскрывается в

```
extern "C" HRESULT __stdcall
```

С помощью программы REGSVR32.EXE эти функции можно вызвать для регистрации компонента. Эта вездесущая утилита, вероятно, уже есть на Вашем компьютере. Большинство программ установки вызывают *DllRegisterServer* в процессе своей работы. Для этого нужно просто загрузить DLL с помощью *LoadLibrary*, получить адрес функции с помощью *GetProcAddress* и потом, наконец, вызвать функцию.

Реализация DllRegisterServer

Реализация *DllRegisterServer* — простой код обновления Реестра. Win32 содержит множество функций, добавляющих и удаляющих разделы Реестра. Для регистрации наших компонентов и удаления их из Реестра понадобятся только шесть функций:

```
RegOpenKeyEx
RegCreateKeyEx
RegSetValueEx
RegEnumKeyEx
RegDeleteKey
RegCloseKey
```

Об этих функциях много написано в других книгах, поэтому я не собираюсь детально рассматривать их здесь. Чтобы использовать эти функции, включите в Ваш исходный файл WINREG.H и WINDOWS.H и скомпонуйте программу с ADVAPI32.LIB. Увидеть эти функции в действии Вы сможете в файлах REGISTRY.H и REGISTRY.CPP из примера следующей, восьмой главы.

Категории компонентов

Минималистский взгляд на Реестр Windows состоит в том, что это длинный список CLSID, с каждым из которых связано имя файла. Клиент может просмотреть эти CLSID и выбрать подходящий компонент. Но как клиент определяет, какой именно компонент следует использовать? Один из вариантов — вывести список дружественных имен компонентов и предоставить пользователю выбирать. Однако пользователю вряд ли понравится, если, выбрав компонент из длинного списка, он в итоге узнает, что тот не работает. Загружать каждый из компонентов, упомянутых в Реестре, и запрашивать у них необходимые нам интерфейсы — слишком затяжное мероприятие. Надо как-то уметь определять, поддерживает ли компонент нужные интерфейсы, до создания экземпляра этого компонента.

Решение этой проблемы дают *категории компонентов (component categories)*. Категория компонентов — это набор интерфейсов, которым присвоен CLSID, называемый в данном случае CATID. Компоненты, реализующие все интерфейсы некоторой категории, могут зарегистрироваться как члены данной категории. Это позволяет клиентам более осмысленно выбирать компоненты из реестра, рассматривая только те, которые принадлежат к некоторой категории.

Категория компонентов — тоже своего рода договор между компонентом и клиентом. Регистрируя себя в некоторой категории, компонент тем самым гарантирует, что поддерживает все входящие в категорию интерфейсы. Категории могут использоваться для типизации компонентов. использование категорий аналогично использованию абстрактных базовых классов в C++. Абстрактный базовый класс — это набор функций, которые производный класс обязан реализовать; поэтому можно сказать, что производный класс — конкретная реализация данного абстрактного базового класса. Категория компонентов — набор интерфейсов, которые должны быть реализованы компонентом, чтобы тот относился к данной категории. Компонент, принадлежащий категории, — конкретная реализация данной категории.

Компонент может входить в произвольное число категорий. Компонент не обязан поддерживать исключительно те интерфейсы, которые определены категорией; он может поддерживать любой интерфейс в дополнение к ним.

Одно из применений категорий — задание набора интерфейсов, которые компонент обязан поддерживать. Альтернативой служит задание набора интерфейсов, которые компонент требует от своего клиента. Компоненту для нормальной работы могут потребоваться от клиента некоторые сервисы. Например, трехмерному графическому объекту для работы может потребоваться определенная графическая библиотека (graphic engine).

Реализация категорий компонентов

Самое приятное в категориях компонентов то, что для их использования Вам не нужно возиться с Реестром самостоятельно. В системах Windows имеется стандартный Диспетчер категорий компонентов (Component Category Manager), который проделает за Вас всю работу. Этот Диспетчер (CLSID_StdComponentCategoryMgr) — стандартный компонент COM, реализующий два интерфейса, *ICatRegister* и *ICatInformation*. *IcatRegister* используется для регистрации и удаления категорий. Он также может

использоваться для добавления и удаления компонентов к категории. *ICatInformation* применяется для получения информации о категориях в системе. С помощью этого интерфейса Вы можете найти:

- все категории, зарегистрированные в системе;
- все компоненты, принадлежащие данной категории;
- все категории, к которым принадлежит данный компонент.

С помощью Диспетчера категорий легко добавлять и удалять категории. Использование Диспетчера показано в примере программы из этой главы, который можно найти на диске. Программа выдает список зарегистрированных в системе категорий компонентов, добавляет новую категорию и, наконец, удаляет эту категорию. Если у Вас эта программа не работает, то возможно, что на Вашем компьютере не установлены некоторые файлы. В прилагаемом к примеру файле README указаны файлы, которые могут отсутствовать, и поясняется, как их установить.

Даже если Вам не нужны категории компонентов, данный пример все равно представляет интерес, так как это первый случай использования нами компонента COM, реализованного кем-то другим.

OleView

Редактор Реестра показывает Реестр «в чистом виде», что полезно для изучения. Вы должны знать организацию Реестра, чтобы реализовать самостоятельно регистрирующиеся компоненты или клиентов, которые будут опрашивать Реестр. Однако если Вам нужна дополнительная информация об установленных на компьютере компонентах, использование Редактора Реестра может потребовать слишком много времени. Ведь он, по существу, показывает все данные в виде списка CLSID.

Другая программа из Win32 SDK — OleView — представляет информацию на более высоком уровне. Вместо длинного списка CLSID и других GUID OleView отображает деревья, содержащие элементы с дружественными именами. Кроме того, OleView позволяет просматривать категории компонентов, установленных в системе. Для изучения лучше всего запустить OleView и поработать. Я использовал эту программу для проверки моего кода саморегистрации. Если OleView может найти информацию, то, скорее всего, эта информация помещена в правильное место.

Некоторые функции библиотеки COM

Всем клиентам и компонентам COM приходится выполнять много типовых операций. Чтобы сделать выполнение этих операций стандартным и совместимым, COM предоставляет библиотеку функций. Библиотека реализована в OLE32.DLL. Для статической компоновки с ней Вы можете использовать OLE32.LIB. В этом разделе мы рассмотрим некоторые из важных типовых операций.

Инициализация библиотеки COM

Во-первых, рассмотрим инициализацию самой библиотеки COM. Процесс должен вызвать *CoInitialize* для инициализации библиотеки, прежде чем использовать ее функции (за исключением функции *CoBuildVersion*, возвращающей номер версии библиотеки). Когда процесс завершает работу с библиотекой COM, он должен вызвать *CoUninitialize*.

Прототипы этих функций приведены ниже:

```
HRESULT CoInitialize(void* reserved); // Значение параметра должно быть NULL
void CoUninitialize();
```

Библиотека COM требует инициализации только один раз для каждого процесса. Много кратные вызовы процессом *CoInitialize* допустимы, но каждому из них должен соответствовать отдельный вызов *CoUninitialize*. Если *CoInitilialize* уже была вызвана данным процессом, то она возвращает не S_OK, а S_FALSE. Поскольку в данном процессе библиотеку COM достаточно инициализировать лишь один раз, и поскольку эта библиотека используется для создания компонентов, компонентам в процессе не требуется инициализировать библиотеку. По общему соглашениюCOM инициализируется в EXE, а не в DLL.

Использование *OleInitialize*

OLE, построенная «поверх» COM, добавляет поддержку библиотек типов, буфера обмена, перетаскивания мышью, документов ActiveX, Автоматизации и управляющих элементов ActiveX. Библиотека OLE содержит дополнительную поддержку этих возможностей. Если Вы хотите использовать все это, следует вызывать *OleInitialize* и *OleUninitialize* вместо *CoInitialize* и *CoUninitialize*. Обычно проще всего вызвать функции *Ole** и забыть о них. Функции *Ole** вызывают соответствующие функции *Com**. Однако использование *Ole** вместо *Com** приводит к излишним расходам ресурсов и времени, если расширенные возможности не используются.

CoInitializeEx

В операционных системах Windows, поддерживающих DCOM, Вы можете использовать *CoInitializeEx*, чтобы пометить компонент как использующий модель свободных потоков (free-threaded). Более подробная информация о *CoInitializeEx* содержится в гл. 13.

Управление памятью

Очень часто внутренняя функция компонента выделяет блок памяти, который возвращается клиенту через выходной параметр. Но кто и каким образом будет эту память освобождать? Наибольшая проблема связана с тем, кто освободит память, — ведь клиент и компонент могут быть реализованы разными людьми, написаны на разных языках и даже выполняться в разных процессах. Необходим стандартный способ выделения и освобождения такой памяти.

Решение предоставляет менеджер памяти задачи (task memory allocator) COM. С его помощью компонент может передать клиенту блок памяти, который тот будет в состоянии освободить. Кроме того, менеджер «гладко» работает с потоками, поэтому его можно применять в многопоточных приложениях.

Как обычно, менеджер используется через интерфейс. В данном случае интерфейс называется *Imalloc* и возвращается функцией *CoGetMalloc*. *IMalloc::Alloc* выделяет блок памяти, а *IMalloc::Free* освобождает память, выделенную с помощью *IMalloc::Alloc*. Однако обычно вызывать *CoGetMalloc* для получения указателя на интерфейс, вызывать с помощью этого указателя функцию и затем освобождать указатель — значит делать слишком много работы. Поэтому библиотека COM предоставляет удобные вспомогательные функции — *CoTaskMemAlloc* и *CoTaskMemFree*:

```
void* CoTaskMemAlloc(
  ULONG cb // Размер выделяемого блока в байтах
};

void CoTaskMemFree(
  void* pv // Указатель на освобождаемый блок памяти
};
```

Память, выделенную и переданную при помощи выходного параметра, всегда освобождает вызывающая процедура (пользующаяся *CoTaskMemFree*).

Преобразование строк в GUID

В Реестре содержатся строковые представления CLSID. Поэтому нам нужны специальные функции для преобразования CLSID в строку и обратно. В библиотеке COM имеется несколько удобных функций такого рода.
StringFromGUID2 конвертирует GUID в строку:

```
wchar_t szCLSID[39];
int r = ::StringFromGRUID2(CLSID_Component1, szCLSID, 39);
```

StringFromGUID2 генерирует строку символов Unicode, т.е. строку двухбайтовых символов типа *wchar_t*, а не *char*. В системах, не использующих Unicode, Вам придется преобразовать результат в *char*. Для этого можно прибегнуть к функции ANSI *wcstombs*, как показано ниже.

```
#ifndef _UNICODE
  // Преобразование из строки Unicode в обычную
  char szCLSID_single[39];
  wcstombs(szCLSID_single, szCLSID, 39);
#endif
```

Есть еще несколько функций, выполняющих аналогичные операции:

Функция	Назначение
StringFromCLSID	Безопасное с точки зрения приведения типов преобразование CLSID в строку
StringFromIID	Безопасное с точки зрения приведения типов преобразование IID в строку
StringFromGUID2	Преобразование GUID в текстовую строку; строка возвращается в буфер, выделенный вызывающей программой
CLSIDFromString	Безопасное с точки зрения приведения типов преобразование строки в CLSID
IIDFromString	Безопасное с точки зрения приведения типов преобразование строки в IID

Некоторые их этих функций требуют использовать менеджер памяти задачи из предыдущего раздела:

```
wchar_t* string;
```

```
// Получить строку из CLSID
::StringFromCLSID(CLSID_Component1, &string);

// Использовать строку
...

// Освободить строку
::CoTaskMemFree(string);
```

Резюме

Строим ли мы дома (например, в гостиной) самолет, пишем ли ночами книгу или разрабатываем компоненты, большая часть нашего времени и энергии уходит на тысячи деталей. Внимание к деталям и правильное обращение с ними определяет успех. В этой главе Вы узнали, что COM использует HRESULT для возвращения кодов успеха или ошибки. Вы узнали о GUID — удивительной структуре данных, которая основана на алгоритме, позволяющем кому угодно, где угодно и когда угодно получать уникальный идентификатор. Вы также видели, что COM использует GUID для идентификации практически всех объектов, в том числе компонентов (CLSID) и интерфейсов (IID).

Вы узнали и о том, как CLSID транслируется в имя файла компонента с помощью Реестра Windows. Для регистрации компонента программа установки или REGSVR32.EXE вызывает функцию *DllRegisterServer*, экспортированную DLL компонента. В минимальном варианте компонент помещает в Реестр свой CLSID и имя файла.

В следующей главе мы увидим, как COM создает компонент при помощи CLSID. Это гораздо проще, чем построить самолет в гостиной.

Замечание о макросах определения интерфейсов

Существуют макросы, облегчающие программистам переход с C на C++; они помогают добиться того, чтобы одно и то же определение интерфейса работало в программах на обоих языках. Эти макросы есть как в OBJBASE.H, так и в BASETYPS.H. Ранее в примерах я использовал следующий простой интерфейс:

```
interface IX : IUnknown
{
  virtual void __stdcall Fx() = 0;
};
```

При использовании упомянутых макросов этот интерфейс выглядит так:

```
DECLARE_INTERFACE(IX, IUnknown)
{
  // IUnknown
  STDMETHOD(QueryInterface) (THIS_ REFID, PPVOID) PURE;
  STDMETHOD_(ULONG, AddRef) (THIS) PURE;
  STDMETHOD_(ULONG, Release) (THIS) PURE;
  // IX
  STDMETHOD_(void, Fx) (THIS) PURE;
}
```

Однако сам я не использую эти макросы, предпочитая писать код так, чтобы он выглядел и работал, как код на C++. Если бы я собирался публиковать свои компоненты, чтобы их использовали другие люди, то писал бы интерфейс на специальном языке. Этот язык описания интерфейсов, называемый IDL, рассматривается в гл. 11 и 12.

8. Фабрика класса

Когда я был совсем маленьким и еще не собирался стать пожарным, я мечтал стать дизайнером наборов конструктора Lego. У меня были самые разные идеи относительно хитроумных новых деталей, из которых можно было бы строить потрясающие модели. Я даже послал несколько проектов в компанию (которая не стала запускать их в производство). Тем не менее, несмотря на отсутствие у фирмы интереса к моим новациям, сейчас я мог бы производить детали Lego прямо у себя в спальне.

Уже появились машинки, которые называют *трехмерными принтерами (3D-printers)*, — и это название очень им подходит. Они похожи на струйные принтеры, но выбрасывают тонкую струю пластика под давлением, а не чернила. Такой принтер наносит пластмассу слоями тоньше миллиметра. Повторная «печать» по одному и тому же месту позволяет создать сложные трехмерные объекты. Их можно использовать как прототипы или формы для изготовления деталей, а иногда и как готовые детали. С такой машинкой можно было бы организовать Домашнюю Фабрику Пластиковых Деталей. При помощи пакета САПР можно было бы в мгновение ока проектировать и производить новые детали. На такой домашней фабрике Вы могли бы сделать ту хитрую детальку с переходом 1x3, без которой никак не собиралась вся модель. Вообще у Вас больше не было бы недостатка в деталях — хотя компания Lego, вероятно, предпочла бы все же продавать Вам свои.

В этой главе я собираюсь рассмотреть своего рода фабрику, на которой производятся не детали Lego, а компоненты. Эта фабрика класса — просто компонент с интерфейсом для создания других компонентов, так что обойдется она нам дешевле, чем трехмерный принтер за 50000 долларов.

Но прежде чем заняться фабрикой класса, мы познакомимся с самым простым способом создания компонентов — при помощи функции *CoCreateInstance*. Не удивительно, что этим способом пользуются чаще всего. К сожалению, он недостаточно гибок и годится не для всех компонентов. Все компоненты создаются на этой фабрике — *CoCreateInstance* при создании компонента тоже пользуется ее услугами, но неявно и незаметно для вызывающей программы. Клиент получает большую свободу в создании компонентов, если он прямо использует фабрику. Точно так же, как у Вас было бы больше возможностей, если бы Вы не покупали детали Lego у фирмы, а делали их сами, на Домашней Фабрике Пластиковых Деталей.

CoCreateInstance

Для создания компонентов в библиотеке COM служит функция *CoCreateInstance*, которая, получив CLSID, создает экземпляр соответствующего компонента и возвращает интерфейс этого экземпляра. В этом разделе мы рассмотрим использование *CoCreateInstance* и увидим, с какими ограничениями оно связано. Но сначала давайте посмотрим на саму функцию.

Прототип *CoCreateInstance*

Объявление *CoCreateInstance* приведено ниже:

```
HRESULT __stdcall CoCreateInstance(
  const CLSID& clsid,
  IUnknown* pUnknownOuter, // Внешний компонент
  DWORD dwClsContext, // Контекст сервера
```

```
  const IID& iid,
  void** ppv
);
```

У функции четыре входных параметра (in) и единственный выходной (out). Первый параметр — CLSID создаваемого компонента. Второй параметр используется для агрегирования компонентов и будет обсуждаться в следующей главе. Третий параметр — *dwClsContext* — ограничивает контекст исполнения компонента, с которым данный клиент может работать. Этот параметр мы рассмотрим позже. Четвертый параметр, *iid* — это IID интерфейса, который мы хотим использовать для работы с компонентом. Указатель на этот интерфейс возвращается через последний параметр — *ppv*. Поскольку в *CoCreateInstance* передается IID, клиент может не вызывать *QueryInterface* для созданного компонента.

Использование *CoCreateInstance*

Используется *CoCreateInstance* так же просто, как и *QueryInterface*:

```
// Создать компонент
IX* pIX = NULL;

HRESULT hr = ::CoCreateInstance(
      CLSID_Component1,
      NULL,
      CLSCTX_INPROC_SERVER,
      IID_IX,
      (void**)&pIX
      );

if (SUCCEEDED(hr))
{
  pIX->Fx();
  pIX->Release();
};
```

В данном примере мы создаем компонент, задаваемый *CLSID_Component1*. Мы не агрегируем его, поэтому значением второго параметра является NULL. В следующей главе мы будем передавать функции значение, отличное от NULL. Параметр CLSCTX_INPROC_SERVER заставляет *CoCreateInstance* загружать только те компоненты, которые содержатся в серверах в процесса или в DLL.

Значения, передаваемые *CoCreateInstance* в качестве двух последних параметров, — те же самые, которые мы передавали бы *QueryInterface*. В данном примере мы передаем *IID_IX*, чтобы запросить интерфейс *IX*, который возвращается указателем *pIX*. Если вызов *CoCreateInstance* был успешным, то интерфейс *IX* готов к работе. Освобождение же интерфейса *IX* указывает, что клиент завершил использование и этого интерфейса, и самого компонента.

Контекст класса

Третий параметр *CoCreateInstance* — *dwClsContext* — используется для управления тем, где может исполняться компонент: в том же процессе, что и клиент, в другом процессе или на другой машине. Значением параметра может быть комбинация признаков, приведенных ниже:

CLSCTX_INPROC_SERVER	- Клиент принимает только компоненты, которые исполняются в одном с ним процессе. Подобные компоненты должны быть реализованы в DLL.

CLSCTX_INPROC_HANDLER - Клиент будет работать с обработчиками в процессе. Обработчик в процессе — это компонент внутри процесса, который реализует только часть компонента. Другие части реализуются компонентом вне процесса — локальным или удаленным сервером.

CLSCTX_LOCAL_SERVER - Клиент будет работать с компонентами, которые выполняются в другом процесса, но на той же самой машине. Локальные серверы реализуются в EXE, как мы увидим в гл. 11.

CLSCTX_REMOTE_SERVER - Клиент допускает компоненты, выполняющиеся на другой машине. Использование этого флага требует задействования DCOM. Мы рассмотрим его в гл. 11.

Один и тот же компонент может быть доступен во всех трех контекстах: удаленном, локальном и в процессе. В некоторых случаях клиент может пожелать использовать только компоненты в процессе, так как они работают быстрее. В других случаях он может не захотеть использоватькомпоненты, работающие в его собственном процессе, так как они имеют доступ к любому участку памяти процесса, что не слишком безопасно. Однако в большинстве случаев клиента не интересует контекст, в котором исполняется компонент. Поэтому в OBJBASE.H определены удобные константы, которые комбинируют (с помощью побитового ИЛИ) приведенные выше значения (см. табл. 8.1).

Значение CLSCTX_REMOTE_SERVER добавляется к CLSCTX_ALL и CLSCTX_SERVER, только если перед включением OBJBASE.H Вы определили символ препроцессора _WIN32_WINNT большим или равным 0x0400. (Тот же эффект даст определение перед включением OBJBASE.H символа препроцессора _WIN32_DCOM.) Предупреждение: если Вы передадите функции *CoCreateInstance* значение CLSCTX_REMOTE_SERVER на системе, которая не поддерживает DCOM, то *CoCreateInstance* возвратит ошибку E_INVALIDARG. Это легко может случиться, если Вы компилируете свою программу с _WIN32_WINNT, большим или равным 0x0400, и затем запускаете ее в системе Microsoft Windows NT 3.51 или Microsoft Windows 95, которые не поддерживают DCOM. Более подробно CLSCTX_LOCAL_SERVER и CLSCTX_REMOTE_SERVER рассматриваются в гл. 11.

Таблица 8.1 *Предопределенные комбинации признаков контекста исполнения*

Константы	**Значения**
CLSCTX_INPROC	CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER
CLSCTX_ALL	CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER
CLSCTX_SERVER	CLSCTX_INPROC_SERVER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER

Листинг кода клиента

В качестве примера в этой главе мы создадим наши первые настоящие клиент и компонент COM. Копии всех исходных файлов находятся на прилагающемся диске.

Листинг 8.1 содержит код клиента. Единственным его существенным отличием от клиентов в гл. 6 является создание компонента с помощью *CoCreateInstance*. Среди других особенностей можно назвать использование *CoInitialize* и *CoUninitialize* для инициализации библиотеки COM (как обсуждается в гл. 7).

CLIENT.CPP

```
//
// Client.cpp - реализация клиента
//

#include <iostream.h>
#include <objbase.h>
#include "Iface.h"

void trace(const char* msg) { cout << "Клиент: \t\t" << msg << endl; }

//
// функция main
//

int main()
{
  // Инициализация библиотеки COM
  CoInitialize(NULL);
  trace("Вызвать CoCreateInstance для создания");
  trace(" компонента и получения интерфейса IX");
  IX* pIX = NULL;
  HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                                  NULL,
                                  CLSCTX_INPROC_SERVER,
                                  IID_IX,
                                  (void**)&pIX);
  if (SUCCEEDED(hr))
  {
    trace("IX получен успешно");
    pIX->Fx(); // Использовать интерфейс IX

    trace("Запросить интерфейс IY");
    IY* pIY = NULL;
    hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
    if (SUCCEEDED(hr))
    {
      trace("IY получен успешно");
      pIY->Fy(); // Использовать интерфейса IY
      pIY->Release();
      trace("Освободить интерфейс IY");
    }
    else
    {
      trace("Не могу получить интерфейс IY");
    }

    trace("Запросить интерфейс IZ");
    IZ* pIZ = NULL;
    hr = pIX->QueryInterface(IID_IZ, (void**)&pIZ);
        if (SUCCEEDED(hr))
    {
      trace("Интерфейс IZ получен успешно");
      pIZ->Fz();
      pIZ->Release();
      trace("Освободить интерфейс IZ");
    }
```

```
    else
    {
      trace("Не могу получить интерфейс IZ");
    }
    trace("Освободить интерфейс IX");
    pIX->Release();
  }
  else
  {
    cout << "Клиент: \t\tНе могу создать компонент. hr = "
         << hex << hr << endl;
   }

  // Закрыть библиотеку COM
CoUninitialize();
return 0;
```

Листинг 8.1 *Полный код клиента*

Но *CoCreateInstance* недостаточно гибка

Создание объектов — очень важная операция в любой объектно-ориентированной системе. Прежде чем использовать объекты, их необходимо создать. Если каждый объект создается по-своему, то будет трудно использовать разные объекты полиморфно. Следовательно, желательно, чтобы создание объектов было как можно более гибким — а все компоненты, в свою очередь, создавались сходным образом. Чем гибче процесс создания, тем легче его адаптировать к нуждам множества компонентов.

Выше мы видели, что *CoCreateInstance* получает CLSID, создает соответствующий компонент и возвращает требуемый указатель на интерфейс. В большинстве случаев *CoCreateInstance* вполне достаточно. Однако *CoCreateInstance* недостаточно гибка, чтобы предоставить клиенту способ управления процессом создания компонента. Когда *CoCreateInstance* возвращает управление, компонент уже создан. Уже поздно управлять тем, где он загружается в память, или проверять, есть ли вообще у клиента право создавать компонент. Проблема состоит в том, как управлять созданием компонента. Нам не нужно беспокоиться об управлении инициализацией компонента. Компонент нетрудно инициализировать через интерфейс, который можно запросить после создания. Но Вы не можете получить интерфейс компонента до тех пор, пока тот не создан, — а тогда уже поздно задавать условия создания.

Решение состоит в том, чтобы явно использовать другой компонент, единственным назначением которого будет создание нужного нам компонента.

Фабрики класса

На самом деле *CoCreateInstance* не создает компоненты непосредственно. Вместо этого она создает компонент, называемый *фабрикой класса (class factory)*, который затем и порождает нужный компонент. Фабрика класса — это компонент, единственной задачей которого является создание других компонентов. Точнее, конкретная фабрика класса создает компоненты, соответствующие одному конкретному CLSID. Клиент использует поддерживаемые фабрикой класса интерфейсы для управления тем, как фабрика создает каждый компонент. Стандартный интерфейс создания компонентов — *IClassFactory*. Компоненты, создаваемые *CoCreateInstance*, порождаются именно при помощи этого интерфейса.

Теперь давайте посмотрим, как клиент может создать компонент, напрямую используя фабрику класса. Первый шаг — создание самой фабрики. Когда фабрика класса

создана, мы используем некоторый интерфейс, подобный *IClassFactory*, для окончательного создания своего компонента.

Использование *CoGetClassObject*

CoCreateInstance получает CLSID и возвращает указатель на интерфейс компонента. Нам нужна эквивалентная функция, которая по CLSID возвращает указатель на интерфейс, принадлежащий фабрике класса данного CLSID. Такая функция в библиотеке COM существует и называется *CoGetClassObject*.

Объявление *CoGetClassObject* показано ниже:

```
HRESULT __stdcall CoGetClassObject(
        const CLSID& clsid,
        DWORD dwClsContext,
        COSERVERINFO* pServerInfo,    // Зарезервировано для DCOM
        const IID& iid,
        void** ppv
);
```

Как видите, *CoGetClassObject* очень похожа на *CoCreateInstance*. Первым параметром обеих функций является CLSID нужного компонента. Обеим также передается контекст выполнения — *dwClsContext*. Два последних параметра тоже совпадают. Но *CoGetClassObject* возвращает запрашиваемый указатель для фабрики класса, тогда как *CoCreateInstance* — для самого компонента. Отличаются эти функции только одним параметром. *CoCreateInstance* принимает указатель *IUnknown*, тогда как *CoGetClassObject* — указатель на COSERVERINFO. COSERVERINFO используется DCOM для управления доступом к удаленным компонентам. Мы рассмотрим эту структуру в гл. 11. Повторю, существенное различие между *CoGetClassObject* и *CoCreateInstance* состоит в том, что первая возвращает указатель, относящийся к фабрике класса нужного нам компонента, а не к самому компоненту. Требумый компонента создается при помощи интерфейса, указатель на который возвращает *CoGetClassObject*. Обычно это указатель *IClassFactory*.

IClassFactory

Стандартный интерфейс, который поддерживают фабрики класса для создания компонентов, — это*IClassFactory*. Объявление этого интерфейса приводится ниже:

```
interface IClassFactory : IUnknown
{
  HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
  const IID& iid,
  void** ppv);
  HRESULT __stdcall LockServer(BOOL bLock);
}
```

У *IClassFactory* есть две функции-члена, *CreateInstance* и *LockServer*. Обсуждение *LockServer* мы отложим до конца главы.

CreateInstance

IClassFactory::CreateInstance — это еще одна функция со знакомыми параметрами. Первый параметр,*pUnknownOuter* — указатель на интерфейс *IUnknown*. Это тот же самый

указатель, что передается *CoCreateInstance*. Его мы рассмотрим в следующей главе при обсуждении агрегирования компонентов.

Два оставшихся параметра — те же, что у *QueryInterface*. С их помощью вызывающая функция может запросить интерфейс компонента одновременно с созданием последнего. Это экономитклиенту один вызов функции. Если компонент к тому же выполняется на удаленной машине, то так экономится и один цикл запрос-ответ по сети. Самое интересное в *CreateInstance* — не те параметры, которые у нее есть, а параметр, которого у нее нет. *IClassFactory::CreateInstance* не получает в качестве параметра CLSID. Это означает, что данная функция может создавать компоненты, соответствующие только одному CLSID — тому, который был передан *CoGetClassObject*.

IClassFactory2

Microsoft уже объявила еще один интерфейс создания компонентов, дополняющий *IClassFactory*. *IClassFactory2* добавляет к *IClassFactory* поддержку лицензирования или разрешения на создание. Клиент обязан передать фабрике класса при помощи *IClassFactory2* корректный ключ или лицензию, прежде чем та будет создавать компоненты. С помощью *IClassFactory2* фабрика класса может гарантировать, что клиент получил компонент легально и имеет право пользования. Я уверен, что это не последний интерфейс создания компонентов.

CoCreateInstance vs. *CoGetClassObject*

Создание фабрики класса, получение указателя *IClassFactory* и затем создание компонента — это большая работа, которую приходится проделывать всякий раз при создании компонента. Вот почему в большинстве программ используется *CoCreateInstance*, а не *CoGetClassObject*. Однако, как я уже говорил, *CoCreateInstance* в действительности реализована при помощи *CoGetClassObject*. Ниже приведен код, показывающий, как это могло бы быть сделано.

```
HRESULT CoCreateInstance(const CLSID& clsid,
        IUnknown* pUnknownOuter,
        DWORD dwClsContext,
        const IID& iid,
        void** ppv)
{
  // Установить в NULL выходной параметр
  *ppv = NULL;

  // Создать фабрику класса и получить указатель на интерфейс IClassFactory
  IClassFactory* pIFactory = NULL;
  HRESULT hr = CoGetClassObject(clsid,
               dwClsContext,
               NULL,
               IID_IClassFactory,
               (void**)&pIFactory);
  if (SUCCEEDED(hr))
  {
    // Создать компонент
    hr = pIFactory->CreateInstance(pUnknownOuter, iid, ppv);
    // Освободить фабрику класса
    pIFactory->Release();
  }
  return hr;
}
```

CoCreateInstance вызывает *CoGetClassObject* и получает указатель на интерфейс *IClassFactory* фабрики класса. Затем с помощью полученного указателя *CoCreateInstance* вызывает *IClassFactory::CreateInstance*, результатом чего и является создание нового компонента.

Зачем нужна *CoGetClassObject*?

В большинстве случаев можно забыть о *CoGetClassObject* и создать компонент при помощи *CoCreateInstance*. Однако есть два случая, в которых следует использовать именно *CoGetClassObject*, а не *CoCreateInstance*. Во-первых, Вы должны использовать *CoGetClassObject*, если хотите создать объект с помощью интерфейса, отличного от *IClassFactory*. Так, если Вам нужен *IClassFactory2*, придется использовать *CoGetClassObject*. Во-вторых, если Вы хотите сразу создать много компонентов, то эффективнее будет один раз создать фабрику класса для всех компонентов, вместо того, чтобы создавать и освобождать ее для каждого экземпляра компонента. *CoGetClassObject* дает клиенту столь необходимую возможность управления процессом создания.

Создавать компоненты вручную с помощью фабрики класса гораздо хлопотнее, чем позволить *CoCreateInstance* сделать это за Вас. Но если Вы запомните, что фабрика класса — это просто компонент, который создает другие компоненты, будет гораздо легче понять, как это делается.

Фабрики класса инкапсулируют создание компонентов

Мне бы хотелось отметить некоторые характеристики фабрик класса. Прежде чем показать Вам, как их реализовывать. Во-первых, данный экземпляр фабрики класса создает компоненты, соответствующие только одному CLSID. Это очевидно, поскольку *CoGetClassObject* имеет в качестве параметра CLSID, а *IClassFactory::CreateInstance* нет. Во-вторых, фабрика класса для данного CLSID создается тем же разработчиком, который реализует соответствующий компонент. Компонент-фабрика класса в большинстве случаев содержится в той же DLL, что и создаваемый компонент.

Конкретный экземпляр фабрики класса соответствует одному конкретному CLSID, и как фабрика, так и создаваемый ею компонент реализуются одним и тем же программистом. Поэтому фабрика класса может иметь и имеет специфические знания о создаваемом компоненте. Реализация фабрики класса с использованием специфической информации о создаваемом компоненте — отнюдь не программистская небрежность. Задача фабрики класса состоит в том, чтобы знать, как создается компонент, и инкапсулировать это знание так, чтобы максимально изолировать клиент от требований компонента.

Реализация фабрики класса

В этом разделе мы рассмотрим реализацию компонента, обращая особое внимание на реализацию фабрики класса. Но сначала посмотрим, как создаются сами фабрики класса.

Использование *DllGetClassObject*

В гл. 6 функция *CallCreateInstance* вызывала для создания компонента функцию *CreateInstance* из DLL. Функции *CoGetClassObject* также нужна точка входа DLL для создания фабрики класса компонента, которая (фабрика — *ред.*) реализована в одной DLL с компонентом. Эта точка входа называется *DllGetClassObject*. *CoGetClassObject* вызывает

функцию *DllGetClassObject*, которая в действительности создает фабрику класса. *DllGetClassObject* объявлена так:

```
STDAPI DllGetClassObject(
       const CLSID& clsid,
       const IID& iid,
       void** ppv);
```

Три параметра этой функции Вам уже знакомы: это те же параметры, что передаются *CoGetClassObject*. Первый — идентификатор класса компонентов, которые будет создавать фабрика класса. Второй — идентификатор интерфейса фабрики, который желает использовать клиент. Указатель на этот интерфейс возвращается через третий параметр.

Весьма существенно, что *DllGetClassObject* передается CLSID. Этот параметр позволяет одной DLL поддерживать несколько компонентов, так как по значению CLSID можно выбрать подходящую фабрику класса.

Общая картина

Набросок общей картины создания компонента представлен на рис. 8.1. Здесь Вы увидите основных «участников» процесса. Во-первых, это клиент, который инициирует запрос обращением к *CoGetClassObject*. Во-вторых, это библиотека COM, реализующая *CoGetClassObject*. В-третьих, это DLL. DLL содержит функцию *DllGetClassObject*, которая вызывается *CoGetClassObject*. Задача *DllGetClassObject* — создать запрошенную фабрику класса. Способ, которым она это делает, оставлен полностью на усмотрение разработчика, так как он скрыт от клиента.

После того, как фабрика класса создана, клиент использует интерфейс *IClassFactory* для создания компонента. Как именно *IClassFactory::CreateInstance* создает компонент — дело разработчика. Как уже отмечалось, *IClassFactory* инкапсулирует этот процесс, поэтому при создании компонента фабрика класса может использовать специфические знания.

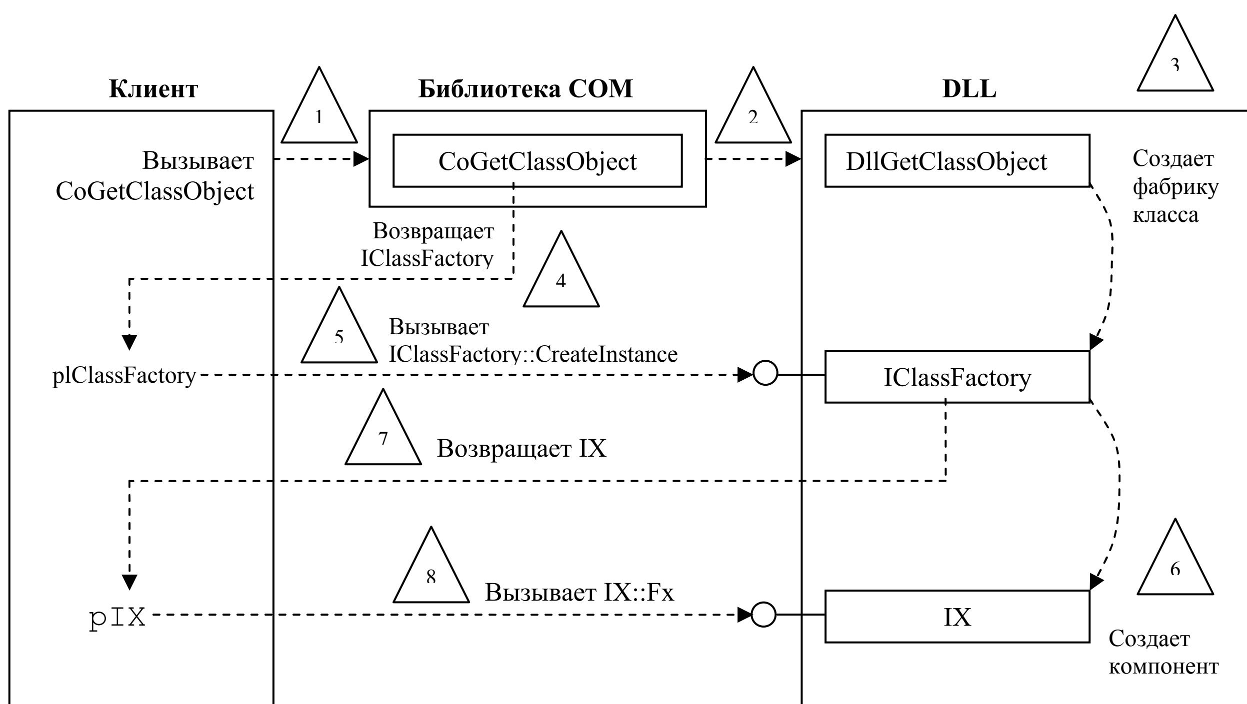


Рис. 8.1 *Пронумерованный точки показывают последовательность создания клиентом компонента с помощью библиотеки COM и фабрики класса*

Теперь мы готовы рассмотреть реализацию компонента.

Листинг кода компонента

Реализация компонента и его фабрики класса показаны в листинге 8.2. Фабрика реализована классом C++ *CFactory*. Первое, что Вы должны заметить в *CFactory* — это просто еще один компонент. Он реализует *IUnknown* так же, как и другие компоненты. Единственное отличие между реализациями *CFactory* и *CA* составляют наборы поддерживаемых интерфейсов. Просматривая код, особое внимание уделите *CFactory::CreateInstance* и *DllGetClassObject*.

CMPNT.CPP

```
//
// Cmpnt.cpp
//
#include <iostream.h>
#include <objbase.h>
#include "Iface.h" // Объявления интерфейсов
#include "Registry.h" // Функции дляработы с Реестром

// Функция трассировки
void trace(const char* msg) { cout << msg << endl; }

///////////////////////////////////////////////////////////
//
// Глобальные переменные
//
static HMODULE g_hModule = NULL; // Описатель модуля DLL
static long g_cComponents = 0; // Количество активных компонентов
static long g_cServerLocks = 0; // Счетчик блокировок
// Дружественное имя компонента
const char g_szFriendlyName[] = "Inside COM, Chapter 7 Example";
// Не зависящий от версии ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap07";
// ProgID
const char g_szProgID[] = "InsideCOM.Chap07.1";
///////////////////////////////////////////////////////////
//
// Компонент
//

class CA : public IX, public IY
{
public:
  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
  // Интерфейс IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }
  // Интерфейс IY
  virtual void __stdcall Fy() { cout << "Fy" << endl; }
  // Конструктор
  CA();
  // Деструктор
  ~CA();
private:
  // Счетчик ссылок
  long m_cRef;
};
```

```
//
// Конструктор
//
CA::CA() : m_cRef(1)
{
  InterlockedIncrement(&g_cComponents);
}

//
// Деструктор
//
CA::~CA()
{
  InterlockedDecrement(&g_cComponents);
  trace("Компонент:\t\tСаморазрушение");
}

//
// Реализация IUnknown
//
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IX)
  {
    *ppv = static_cast<IX*>(this);
    trace("Компонент:\t\tВернуть указатель на IX");
  }
  else if (iid == IID_IY)
  {
    *ppv = static_cast<IY*>(this);
    trace("Компонент:\t\tВернуть указатель на IY");
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }

  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CA::AddRef()
{
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CA::Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}
```

```
///////////////////////////////////////////////////////////
//
// Фабрика класса
//
class CFactory : public IClassFactory
{
public:
  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
  // Интерфейс IClassFactory
  virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
  const IID& iid,
  void** ppv);
  virtual HRESULT __stdcall LockServer(BOOL bLock);
  // Конструктор
  CFactory() : m_cRef(1) {}
  // Деструктор
  ~CFactory() { trace("Фабрика класса:\t\tСаморазрушение"); }
private:
  long m_cRef;
};

//
// Реализация IUnknown для фабрики класса
//
HRESULT __stdcall CFactory::QueryInterface(const IID& iid, void** ppv)
{
  if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
  {
    *ppv = static_cast<IClassFactory*>(this);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CFactory::AddRef()
{
  return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CFactory::Release()
{
  if (InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}
```

```
//
// Реализация IClassFactory
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
const IID& iid,
void** ppv)
{
  trace("Фабрика класса:\t\tСоздать компонент");
  // Агрегирование не поддерживается
  if (pUnknownOuter != NULL)
  {
    return CLASS_E_NOAGGREGATION;
  }
  // Создать компонент
  CA* pA = new CA;
  if (pA == NULL)
  {
    return E_OUTOFMEMORY;
  }
  // Вернуть запрошенный интерфейс
  HRESULT hr = pA->QueryInterface(iid, ppv);
  // Освободить указатель на IUnknown
  // (При ошибке в QueryInterface компонент разрушит сам себя)
  pA->Release();
  return hr;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
  if (bLock)
  {
    InterlockedIncrement(&g_cServerLocks);
  }
  else
  {
    InterlockedDecrement(&g_cServerLocks);
  }
  return S_OK;
}

///////////////////////////////////////////////////////////
//
// Экспортируемые функции
//
//
// Можно ли выгружать DLL?
//
STDAPI DllCanUnloadNow()
{
  if ((g_cComponents == 0) && (g_cServerLocks == 0))
  {
    return S_OK;
  }
  else
  {
    return S_FALSE;
  }
}

//
// Получить фабрику класса
//
```

```
STDAPI DllGetClassObject(const CLSID& clsid,
const IID& iid,
void** ppv)
{
  trace("DllGetClassObject:\tСоздать фабрику класса");
  // Можно ли создать такой компонент?
  if (clsid != CLSID_Component1)
  {
    return CLASS_E_CLASSNOTAVAILABLE;
  }
  // Создать фабрику класса
  CFactory* pFactory = new CFactory; // Счетчик ссылок устанавливается
  // в конструкторе в 1
  if (pFactory == NULL)
  {
    return E_OUTOFMEMORY;
  }
  // Получить требуемый интерфейс
  HRESULT hr = pFactory->QueryInterface(iid, ppv);
  pFactory->Release();
  return hr;
}

//
// Регистрация сервера
//
STDAPI DllRegisterServer()
{
  return RegisterServer(g_hModule,
  CLSID_Component1,
  g_szFriendlyName,
  g_szVerIndProgID,
  g_szProgID);
}

//
// Удаление сервера из Реестра
//
STDAPI DllUnregisterServer()
{
  return UnregisterServer(CLSID_Component1,
  g_szVerIndProgID,
  g_szProgID);
}

////////////////////////////////////////////////////////////
//
// Реализация модуля DLL
//
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    g_hModule = hModule;
  }
  return TRUE;
}
```

Листинг 8.2 *Полный код компонента, фабрики класса и функций, экспортируемых из DLL*

Клиент: Вызвать CoCreateInstance для
Клиент: создания компонента и получения интерфейса IX
DllGetClassObject: Создать фабрику класса
Фабрика класса: Создать компонент
Компонент: Вернуть указатель на IX
Фабрика класса: Саморазрушение
Клиент: IX получен успешно
Fx
Клиент: Запросить интерфейс IX
Компонент: Вернуть указатель на IY
Клиент: IY получен успешно
Fy
Клиент: Освободить интерфейс IY
Клиент: Запросить интерфейс IZ
Клиент: Не могу получить интерфейс IZ
Клиент: Освободить интерфейс IX
Компонент: Саморазрушение

Только что представленная реализация *DllGetClassObject* делает три вещи. Во-первых, она проверяет, соответствует ли запрос именно той фабрике, которую она умеет создавать. Затем при помощи операции *new* создается фабрика класса. Наконец, *DllGetClassObject* запрашивает у фабрики класса интерфейс, требуемый клиенту. Реализация *IClassFactory::CreateInstance* похожа на реализацию *DllGetClassObject*. Обе функции создают компонент и запрашивают у него интерфейс. *IClassFactory::CreateInstance* создает *CA*, тогда как *DllGetClassObject* — *CFactory*.

Обратите внимание, что *DllGetClassObject* обладает подобными знаниями о создаваемой ею фабрике класса, а *IClassFactory::CreateInstance* — о создаваемом компоненте. Эти функции изолируют клиент от деталей реализации компонента. Можно сделать эти функции пригодными для повторного использования, однако в том или ином месте им обязательно потребуются специфические знания о том, как создавать конкретную фабрику класса или конкретный компонент. Способы, используемые *DllGetClassObject* и *IClassFactory::CreateInstance* для создания компонентов, полностью оставлены на усмотрение разработчика. Повторно применимая реализация *DllGetClassObject* и *IClassFactory::CreateInstance* будет представлена в гл. 10.

Последовательность выполнения

Двайте подробно рассмотрим последовательность выполнения кодов клиента и компонента, приведенных в листингах 8.1 и 8.2. На рис. 8.2 эта последовательность представлена графически. Ось времени на рисунке направлена вниз. Пяти основным структурным элементам — клиенту, библиотеке COM, DLL, фабрике класса и компоненту — соответствуют отдельные колонки. С каждым элементом связана вертикальная линия. Сплошная линия означает, что данный элемент был создан и все еще существует. Пунктирная линия показывает, что элемент еще или уже не существует. Прямоугольники, нанесенные поверх линий, соответствуют временам выполнения операций. Горизонтальные линии — это вызовы функций, передающие управление от одного структурного элемента к другому.

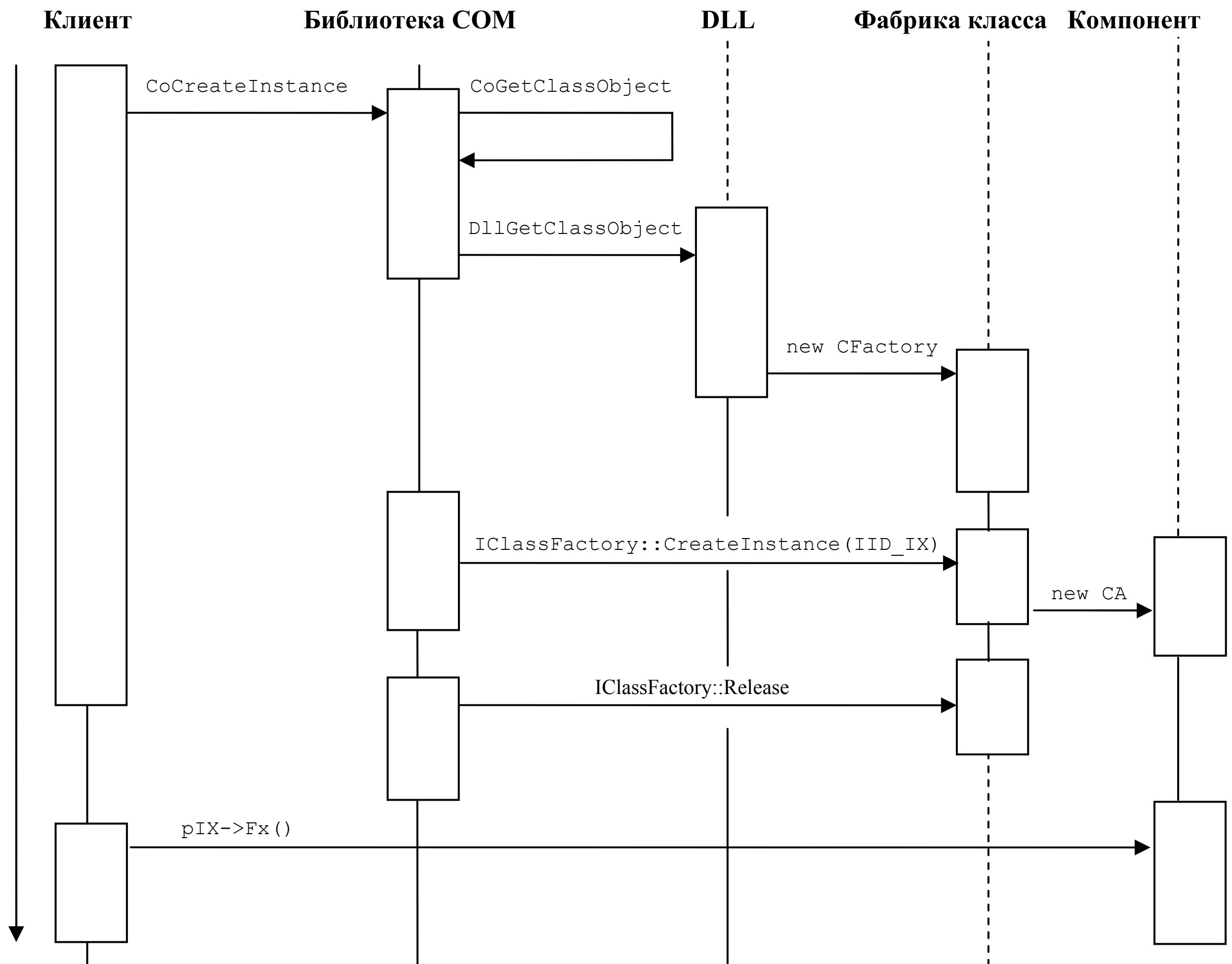


Рис. 8.2 *CoCreateInstance позаботится за клиента о множестве мелких деталей создания компонента*

Для простоты я опустил вызовы членов *IUnknown* и другие мелкие подробности. Кратко рассмотрим содержание рисунка. Сначала клиент вызывает *CoCreateInstance*, которая реализована в библиотеке COM. *CoCreateInstance* реализована с помощью *CoGetClassObject*. *CoGetClassObject* отыскивает компонент в Реестре. Если компонент найден, то *CoGetClassObject* загружает DLL, являющуюся сервером компонента. После загрузки DLL *CoGetClassObject* вызывает *DllGetClassObject*. *DllGetClassObject* реализована DLL-сервером. Ее задача — создать фабрику класса, что делается в данном примере при помощи оператора *new* C++. Кроме того, *DllGetClassObject* запрашивает у фабрики класса интерфейс *IClassFactory*, который возвращается *CoCreateInstance*. Последняя вызывает метод *CreateInstance* этого интерфейса. В нашем примере *IClassFactory::CreateInstance* использует для создания компонента оператор *new*. Кроме того, она запрашивает у компонента интерфейс *IX*. Получив его, *CoCreateInstance* освобождает фабрику класса и возвращает указатель на *IX* клиенту. Затем клиент может использовать данный указатель для вызова методов компонента. Проще некуда.

Регистрация компонента

Компоненты DLL экспортируют четыре функции. Мы уже рассмотрели *DllGetClassObject*, которую функции библиотеки COM используют для создания фабрики класса. Три других экспортируемые функции используются для регистрации компонента.

Функции *DllRegisterServer* и *DllUnregisterServer* регистрируют и удаляют из Реестра Windows информацию о компоненте. Мы кратко рассматривали их в гл. 6. Реализованы эти функции в файле REGISTRY.CPP. Я не буду объяснять их код — он достаточно прост, и при желании Вы сможете разобраться сами. Мы будем использовать тот же файл REGISTRY.H для регистрации компонентов и в последующих главах книги.

Как пояснялось в предыдущей главе, REGSVR32.EXE
вызывает функцию *DllRegisterServer*, т. е. фактически выполняет регистрацию клиента. Если Вы не запускали make-файл, этот шаг Вам придется сделать самостоятельно. Для удобства я привожу командный файл REGISTER.BAT, состоящий из одной этой команды.

DllMain

Чтобы получить имя файла DLL и зарегистрировать его, функции *DllRegisterServer* нужен описатель содержащей ее DLL. Этот описатель передается функции *DllMain*. В программах на С++ есть функция *main*, с которой начинается выполнение. Программы для Windows используют *WinMain*, а DLL — *DllMain*. Реализовать *DllMain* для получения описателя модуля DLL нетрудно:

```
BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReason, void* lpReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    g_hModule = hModule;
  }
  return TRUE;
}
```

Эта функция заносит описатель в глобальную переменную *g_hModule*, откуда его могут считать функции *DllRegisterServer* и *DllUnregisterServer*.

Несколько компонентов в одной DLL

Я уже говорил выше, что *DllGetClassObject* позволяет нам поддерживать несколько компонентов в одной DLL. Ключевой момент здесь — передача *DllGetClassObject* CLSID создаваемого компонента. Для каждого CLSID *DllGetClassObject* легко может создать особую фабрику класса (см. рис. 8.3).

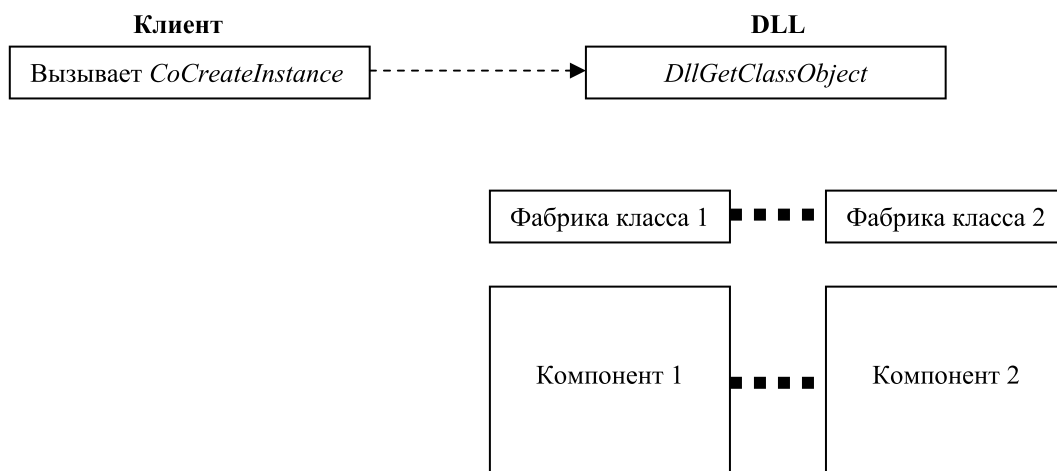


Рис. 8.3 *Одна DLL может содержать несколько компонентов*

Тот факт, что DLL может поддерживать много разных компонентов, — одна из причин, по которым DLL соответствует не компоненту, но серверу компонентов. Концепция DLL, предоставляющей клиенту компоненты по запросу, очень мощна. DLL — это средство распространения реализаций компонентов.

Повторное применение реализации фабрики класса

Если Вы похожи на меня, то Вы терпеть не можете писать один и тот же код снова и снова. Необходимость иметь *CFactory1*, *CFactory2* и *CFactory3*, которые будут снова и снова реализовывать один и тот же код для создания компонентов *CA*, *CB* и *CC*, кажется излишней. В нашем случае в этих разных фабриках класса различались бы только следующие строки:

```
CA* pA = new CA;
pA->QueryInterface(...);
```

Если Вы правильно спроектировали свою фабрику класса и компоненты, то Вам понадобится только одна реализация фабрики для всех компонентов. Я предпочитаю для каждого компонента создавать по одной простой функции. Эта функция создает компонент при помощи операции *new* и возвращает указатель на *IUnknown*. Затем я строю из указателей на эти функции таблицу, индексируемую CLSID каждого компонента. *DllGetClassObject* просто отыскивает в таблице указатель нужной функции, создает фабрику класса и передает ей этот указатель. Затем, вместо того, чтобы непосредственно использовать операцию *new*, фабрика класса при помощи указателя вызывает соответствующую функцию создания компонента (см. рис. 8.4).

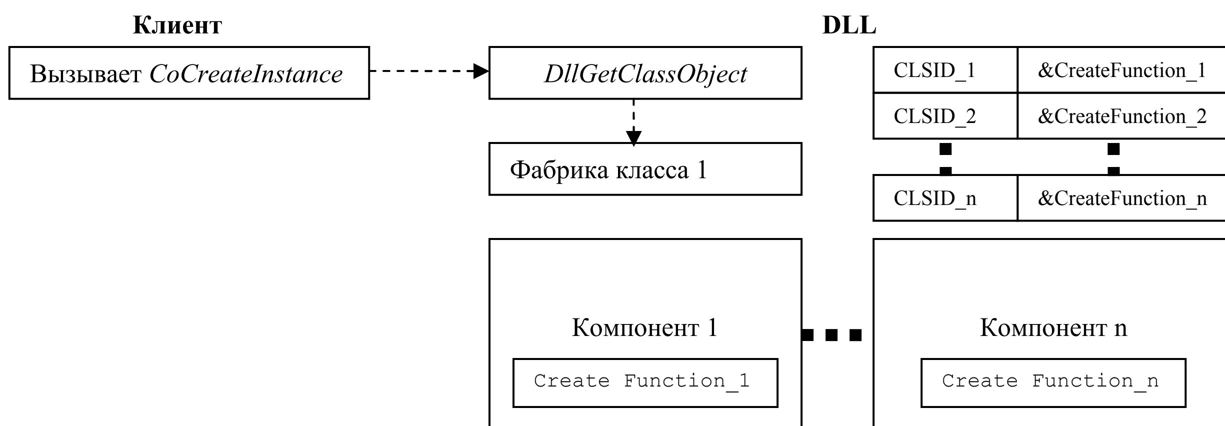


Рис. 8.4 *Простая реализация фабрики класса может обслуживать несколько компонентов*

В гл. 10 мы реализуем повторно применимую фабрику класса, которая использует эту структуру. Прежде чем идти дальше, я хотел бы подчеркнуть один важный момент. Даже если один и тот же код фабрики класса используется несколькими компонентами, данный *экземпляр* фабрики класса может создавать только компоненты, соответствующие одному CLSID. Соотношение между экземплярами фабрики класса и CLSID всегда будет один к одному. Хотя *CFactory* могла бы реализовать все наши классы, любой конкретный экземпляр *CFactory* может создавать только компоненты одного CLSID. Это следствие того, что *IClassFactory::CreateInstance* не передается в качестве параметра CLSID.

Выгрузка DLL

Я достаточно долго избегал разговора о *LockServer* и *DllCanUnloadNow*. Теперь пора заняться ими. Как мы видели в гл. 6, когда клиент динамически связывается с компонентом, он должен загрузить DLL в память. В Win32 для этой цели используется функция *LoadLibrary*1. Когда мы закончили работу с DLL, хотелось бы выгрузить ее из памяти. Не стоит засорять память неиспользуемыми компонентами. Библиотека COM предоставляет функцию *CoFreeUnusedLibraries*, которая, как следует из названия, освобождает неиспользуемые библиотеки. Клиент должен периодически вызывать эту функцию в периоды бездействия.

Использование *DllCanUnloadNow*

Но как *CoFreeUnusedLibraries* определяет, какие из DLL больше не обслуживают ни одного клиента и могут быть выгружены? *CoCreateUnusedLibraries* опрашивает DLL, вызывая *DllCanUnloadNow*. *DllCanUnloadNow* сообщает COM, поддерживает ли данная DLL какие-либо объекты. Если DLL никого не обслуживает, *CoFreeUnusedLibraries* может ее выгрузить. Чтобы проверять, есть ли еще обслуживаемые компоненты, DLL поддерживает их счетчик. Для этого в CMPNT.CPP просто добавляется следующее объявление:

```
static long g_cComponents = 0;
```

Затем *IClassFactory::CreateInstance* или конструктор компонента увеличивают значение *g_cComponents*, а деструктор компонента уменьшает его. *DllCanUnloadNow* дает положительный ответ тогда, когда *g_cComponents* равна 0.

LockServer

Обратите внимание, что я подсчитываю только обслуживаемые DLL в данный момент компоненты, но не фабрики класса. Может оказаться, что логичнее было бы подсчитывать фабрики класса вместе с самими компонентами. Для серверов внутри процесса Вы, если хотите, можете подсчитывать их. Но в гл. 11 мы познакомимся с локальными серверами, которые реализуются в EXE, а не в DLL. С «внутренней» точки зрения, начало и конец работы у серверов внутри и вне процесса отличаются. (С точки зрения «внешней» клиент не видит никаких различий.) Сервер вне процесса запускается таким способом, что внутри него мы не можем подсчитывать фабрики класса, не попадая в порочный круг, в котором сервер никогда не освобождает себя. Более подробно это обсуждается в гл. 11. Достаточно сказать, что присутствие работающей фабрики класса не гарантирует, что сервер будет удерживаться в памяти.

Это создает трудную ситуацию. Выгрузка DLL, у которой имеются исполняющиеся фабрики класса, может вызвать проблемы у клиентов. Предположим, что у клиента имеется указатель на фабрику класса, а соответствующая DLL выгружена. Если клиент попытается использовать указатель на *IClassFactory*, произойдет сбой. Клиенту необходим некоторый способ удержания DLL в памяти, если он собирается использовать указатель *IClassFactory* за пределами одной функции. *IClassFactory::LockServer* позволяет клиенту удерживать сервер в памяти до завершения работы. Клиент просто вызывает *LockServer(TRUE)* для блокирования сервера и *LockServer(FALSE)* для деблокирования.
Реализовать *LockServer* теперь можно простым увеличением и уменьшением счетчика *g_cComponents*. Многие, и я в их числе, предпочитаю использовать отдельные счетчики для компонентов и блокировок. В этом случае *DllCanUploadNow* должна проверять на равенство нулю оба счетчика.

Резюме

В большинстве случаев для создания компонентов пользуются *CoCreateInstance*. Однако иногда эта функция не дает достаточной гибкости. Тогда можно воспользоваться *CoGetClassObject*, которая дает возможность прямо управлять фабрикой класса и интерфейсом, используемым для создания компонентов. Стандартный для создания компонентов интерфейс — *IClassFactory*, который используется и *CoCreateInstance*.

Независимо от того, использует ли клиент *CoCreateInstance* или *CoGetClassObject*, у компонента всегда имеется отдельная фабрика класса. Фабрика класса — это компонент, создающий компонент. Фабрика класса компонента обычно реализует *IClassFactory*. Однако некоторые компонентыпредъявляют особые требования к процессу создания. Такие компоненты вместо (или в дополнение) к *IClassFactory* будут реализовывать и другие интерфейсы.

Если бы мы не могли собирать из деталей Lego конструкции, играть с ними было бы не слишком весело, даже при наличии Домашней Фабрики Пластиковых Деталей. Изготовление деталей — только часть процесса, который описывает эта книга. По-настоящему интересные вещи начинаются тогда, когда мы собираем детали вместе. Так и компоненты становятся по-настоящему интересными, только когда Вы соединяете их вместе, в новую цельную структуру. В следующей главе мы увидим, как построить новые компоненты из уже имеющихся.

9. Повторная применимость компонентов: включение и агрегирование

Авторы статей в компьютерных журналах любят сравнивать COM с самыми разными вещами — например, миллионерами, горохом, C++ и компонентными архитектурами других фирм. Обычно в подобных статьях приводится какая-нибудь таблица вроде этой:

Таблица 9.1 *Гипотетическая таблица из компьютерного журнала*

Свойство	Миллионеры	Горох	C++	COM
Сьедобны	√	√	x	x
Поддерживают наследование	√	√	√	**X**
Могут стать президентом	√	x	x	x

Вы можете не обратить внимания на то, что миллионеры съедобны. Однако, читая эти статьи, Вы не сможете упустить из виду, что COM не поддерживает наследования. Авторы, кажется, не обращают внимания на то, что COM поддерживает полиморфизм — самую важную концепцию объектно-ориентированного программирования, или что COM — небольшая, элегантная и быстро работающая модель, или что компоненты COM могут прозрачно работать по сети. Их не интересует и то, что COM не зависит от языка программирования; что компонентов COM написано больше, чем компонентов какого-либо другого типа. Их интересует только одно модное словцо — наследование!

Поддерживает ли COM наследование? И да, и нет. То, что подразумевается в журнальных статьях — это *наследование реализации*, которое имеет место, когда класс наследует свой код или реализацию от базового класса. Этот вид наследования COM не поддерживается. Однако COM поддерживает *наследование интерфейса*, т.е. наследование классом типа или интерфейса базового класса.

Многие необоснованно утверждают, что COM — плохая технология, так как она не поддерживает наследование реализации. Их аргументация напоминает мне о бурных «войнах» в Интернет — OS/2 против Windows, *vi* против Emacs, Java против Python и т.д. Я не участвую в таких спорах, ибо это пустая трата времени.

COM не поддерживает наследование реализации, потому что наследование реализации слишком тесно привязывает один объект к реализации другого. Если изменится реализация базового объекта, то производные объекты не будут работать, и их тоже нужно будет изменять. Для программы среднего размера на C++ это не очень трудно — у Вас есть доступ ко всем исходным текстам, и Вы можете изменить производные классы. Однако для программ большего размера на изменение всех зависимых классов может уйти недопустимо много времени. Хуже того, у Вас может даже не быть доступа к исходным текстам. Именно поэтому эксперты по созданию больших программ на C++ настоятельно рекомендуют строить приложения на фундаменте абстрактных базовых классов.

Не случайно базовые классы, т.е. род наследования интерфейсов в чистом виде, оказались также и способом реализации интерфейсов COM. Компоненты COM могут быть написаны кем угодно, где угодно и на каком угодно языке. Следовательно, необходимо очень тщательно защищать клиентов компонента от изменений. Наследование реализации не обеспечивает такой защиты.

Поэтому, чтобы гарантировать продолжение работы существующих приложений в случае изменений в компонентах, COM не поддерживает наследование реализации. Но из-за этого не теряются никакие функциональные возможности — ведь наследование

реализации можно целиком смоделировать путем включения компонентов. Конечно, наследование реализации удобнее. Однако я полагаю, что большинство разработчиков предпочтут более устойчивую систему немного более удобной. У наследования реализации своя область применения. В следующей главе я буду использовать его, чтобы избавиться от необходимости реализовывать *IUnknown* для каждого компонента. Но в этой главе мы остановимся на включении компонентов.

Включение и агрегирование

Может быть, это национальная особенность, но, по-моему, мало кто в США доволен существующим положением дел. Мы всегда хотим все улучшить — поэтому постоянно изменяем все подряд, от прически до собственного дома. Так мы поступаем и с компонентами. После того, как кто-то дал Вам компонент, Вы наверняка захотите расширить или подстроить его под свои задачи. Кроме того, Вы можете захотеть использовать новый, усовершенствованный компонент. В C++ подстройка реализуется с помощью включения и наследования. В COM компоненты подстраиваются (специализируются) с помощью *включения (containment)* и *агрегирования (aggregation)*.

Включение и агрегирование — это приемы программирования, в которых один компонент использует другой. Я называю эти два компонента *внешним (outer component)* и *внутренним (inner component)* соответственно. Внешний компонент или *агрегирует*, или *включает* в себя внутренний.

Включение

Включение в COM похоже на включение в C++. Однако, как и все в COM, включение выполняется на уровне интерфейсов. Внешний компонент содержит указатели на интерфейсы внутреннего. Внешний компонент — просто клиент внутреннего компонента. Используя интерфейсы последнего, он реализует свои собственные интерфейсы (рис. 9.1).

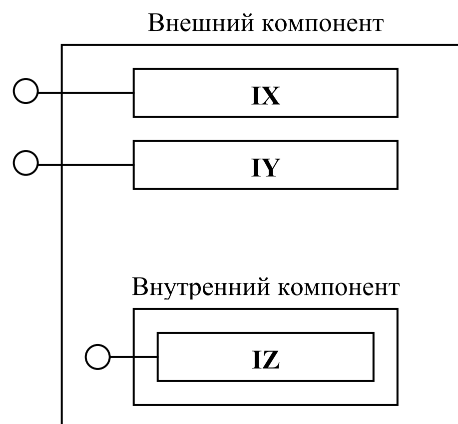


Рис. 9.1 *Внешний компонент содержит внутренний компонент и использует его инетфрейс IZ.*

Внешний компонент может также реализовывать заново интерфейс, поддерживаемый внутренним, передавая последнему вызовы этого интерфейса. Внешний компонент может специализировать этот интерфейс, добавляя свой код перед вызовом внутреннего компонента и после этого (рис. 9.2).

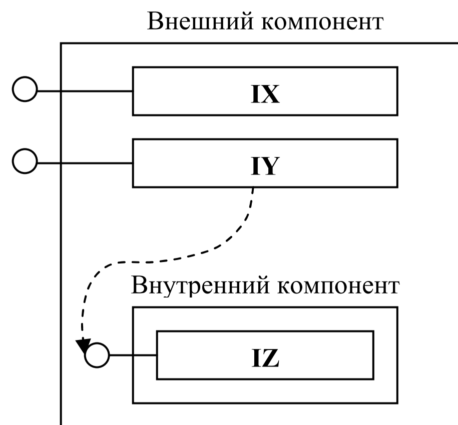


Рис. 9.2 *Внешний компонент содержит внутренний компонент и повторно использует его реалищацию интерфейса IY.*

Агрегирование

Агрегирование — это особый вид включения. Когда внешний компонент агрегирует интерфейс внутреннего компонента, он не реализует интерфейс последнего заново и не передает ему вызовы этого интерфейса явно (как при включении). Вместо этого внешний компонент передает указатель на интерфейс внутреннего компонента непосредственно клиенту. Далее клиент напрямую вызывает методы интерфейса, принадлежащего внутреннему компоненту. При таком подходе внешний компонент избавлен от необходимости реализовывать заново функции интерфейса и передавать вызовы внутреннему компоненту (рис. 9.3). Однако внешний компонент не может специализировать какие-либо функции интерфейса. После того, как внешний компонент передаст интерфейс клиенту, тот обращается к внутреннему компоненту самостоятельно. Клиент не должен знать, что он работает с двумя разными компонентами, так как это нарушит инкапсуляцию. Задача агрегирования — заставить внешний и внутренний компоненты вести себя как один компонент. Как Вы увидите далее, эта возможность достигается при помощи *QueryInterface*.

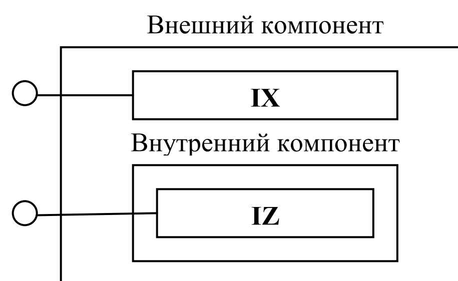


Рис. 9.3 *Когда внешний компонент агрегирует интерфейс, он передает укаатель на него непосредственно клиенту. Он не реализуется интерфейс заново для передачи вызовов внутреннему компоненту.*

Сравнение включения и агрегирования

Небольшой пример пояснит различия между включением и агрегированием. Предположим, что Вы хозяин небольшой металлоремонтной мастерской. У Вас есть две работницы — Памела и Анжела. Памела работает уже давно и знает свое дело досконально. Если заказ будет выполнять Памела, достаточно просто направить клиента

к ней. Анжела, напротив, новичок и у нее нет опыта работы с металлом. Когда работу поручают ей, обычно нужно дать начальные указания, а иногда и сделать вместо нее наиболее сложную часть. Вы же должны и договариваться с клиентом. После того, как Анжела закончит работу, Вы вместе с ней должны проверить результаты и, может быть, что-то подправить. Случай Памелы сходен с подходом агрегирования. Дав работу, Вы уходите со сцены. Однако в случае Анжелы Вы по-прежнему ведете все переговоры с клиентом, даете начальныеуказания и проверяете работу в конце. Этот вариант аналогичен включению.

На каждый агрегируемый Вами интерфейс будут, вероятно, приходиться сотнивключаемых. Помните, всякий раз, когда компонент выступает в роли клиента и использует интерфейс, принадлежащий другому компоненту, первый, в некотором смысле, включает второй. С другой стороны, агрегирование — гораздо более специальный случай. Его используют тогда, когда некоторый компонент уже реализует некоторый интерфейс именно так, как это нужно Вам, и Вы передаете интерфейс компонента клиенту.

Возможны различные сочетания включения и агрегирования. Компонент может специализировать или расширять множество интерфейсов, реализованных многими разными компонентами. Некоторые интерфейсы он может включать, другие же агрегировать. Поскольку агрегирование — это особый случай включения, мы рассмотрим включение первым.

Реализация включения

В этом примере Компонент 1 — внешний; он реализует два интерфейса: *IX* и *IY*. При этом он использует реализацию *IY* Компонентом 2 — внутренним, включаемым компонентом. Это в точности соответствует схеме рис. 9.2.

Клиент и внутренний компонент — практически те же самые, что и клиент с компонентом из предыдущей главы. При включении одного компонента другим ни от клиента, ни от внутреннего компонента не требуется никаких специальных действий. Они даже не знают о самом факте использования включения.

Нам остается рассмотреть только внешний компонент — Компонент 1, который включает Компонент 2. В приведенном ниже листинге 8-1 показано объявление и большая часть реализации Компонента 1. Я выделил полужирным некоторые участки кода, относящиеся к включению. Новая переменная-член *m_pIY* содержит указатель на интерфейс *IY* включаемого Компонента 2.

Код включения из CONTAIN\CMPNT1

```
///////////////////////////////////////////////////////////
//
// Компонент 1
//

class CA : public IX, public IY
{
public:
  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
  // Интерфейс IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }
  // Интерфейс IY
  virtual void __stdcall Fy() { m_pIY->Fy(); }
  // Конструктор
  CA();
```

```
  // Деструктор
  ~CA();
  // Функция инициализации, вызываемая фабрикой класса для
  // создания включаемого компонента
  HRESULT __stdcall Init();
private:
  // Счетчик ссылок
  long m_cRef;
  // Указатель на интерфейс IY включаемого компонента
  IY* m_pIY;
};

//
// Конструктор
//
CA::CA() : m_cRef(1), m_pIY(NULL)
{
  ::InterlockedIncrement(&g_cComponents);
}

//
// Деструктор
//
CA::~CA()
{
  ::InterlockedDecrement(&g_cComponents);
  trace("Самоликвидация");
  // Освободить включаемый компонент
  if (m_pIY != NULL)
  {
    m_pIY->Release();
  }
}

// Инициализация компонента путем создания включаемого компонента
HRESULT __stdcall CA::Init()
{
  trace("Создать включаемый компонент");
  HRESULT hr = ::CoCreateInstance(CLSID_Component2,
                                  NULL,
                                  CLSCTX_INPROC_SERVER,
                                  IID_IY,
                                  (void**)&m_pIY);
  if (FAILED(hr))
  {
    trace("Не могу создать включаемый компонент");
    return E_FAIL;
  }
  else
  {
    return S_OK;
  }
}
```

Листинг 9.1 *Компонент 1 создает экземпляр включаемого компонента и хранит указатель на интерфейс IY последнего*

Давайте рассмотрим, как работает этот код внешнего Компонента 1. Новый метод под названием *Init* создает внутренний Компонент 2 тем же самым способом, которым создают компоненты все клиенты, — посредством вызова *CoCreateInstance*. При этом внешний компонент запрашивает указатель на *IY* у внутреннего и, в случае успеха, сохраняет его в *m_pIY*. В приведенном листинге не показаны реализация *QueryInterface* и

функций внешнего *IUnknown*. Она абсолютно та же, что и в случае, когда включение не используется. Когда клиент запрашивает у Компонента 1 интерфейс *IY*, тот возвращает указатель на свой интерфейс. Затем, когда клиент вызывает метод этого интерфейса, Компонент 1 передает вызов Компоненту 2.
Это выполняет следующая строка:

```
virtual void Fy() { m_pIY->Fy(); }
```

Когда Компонент 1 самоликвидируется, его деструктор вызывает *Release* для указателя *m_pIY*, в результате чего Компонент 2 также удаляет себя.

Фабрика класса Компонента 1 мало изменилась по сравнению с фабрикой класса из предыдущей главы. Единственный новый момент — то, что функция *CreateInstance* вызывает после создания Компонента 1 его функцию *Init*. Код этой функции приведен в листинге 9.2.

Код функции CreateInstance из CONTAIN\CMPNT1

```
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
const IID& iid,
void** ppv)
{
  // Агрегирование не поддерживается
  if (pUnknownOuter != NULL)
  {
    return CLASS_E_NOAGGREGATION;
  }

  // Создать компонент
  CA* pA = new CA;
  if (pA == NULL)
  {
    return E_OUTOFMEMORY;
  }
  // Инициализировать компонент
  HRESULT hr = pA->Init();
  if (FAILED(hr))
  {
    // Ошибка при инициализации. Удалить компонент
    pA->Release();
    return hr;
  }
  // Получить запрошенный интерфейс
  hr = pA->QueryInterface(iid, ppv);
  pA->Release();
  return hr;
}
```

Листинг 9.2 *Фабрика класса внешнего компонента вызывает для вновь созданного компонента функцию Init*

Вот и все, что необходимо для реализации включения. Теперь рассмотрим, для чего включение может применяться.

Расширение интерфейсов

Одно из основных применений включения — расширение интерфейса посредством добавления кода к существующему интерфейсу. Рассмотрим пример. Имеется класс

IAirplane (Самолет), который Вы хотите превратить в *IfloatPlane (Гидросамолет)*. Определения интерфейсов приводятся ниже:

```
interface IAirplane : IUnknown
{
  void TakeOff();
  void Fly();
  void Land();
};

interface IFloatPlane : IAirplane
{
  void LandingSurface(UINT iSurfaceType);
  void Float();
  void Sink();
  void Rust();
  void DrainBankAccount();
};
```

Предположим, что *IAirplane* уже реализован в компоненте MyAirplane. Внешний компонент может просто включить MyAirplane и использовать его интерфейс *IAirplane* для реализации членов *IAirplane*, которые наследует интерфейс *IFloatPlane*:

```
void CmyFloatPlane::Fly()
{
  m_pIAirplane->Fly();
}
```

Другие члены *IAirplane*, вероятно, потребуется модифицировать, чтобы поддерживать взлет и посадку на воду:

```
void CmyFloatPlane::Land()
{
  if (m_iLandingSurface == WATER)
  {
    WaterLanding();
  }
  else
  {
    m_pIAirplane->Land();
  }
}
```

Как видите, использовать включение в данном случае просто. Однако если в интерфейсе *IAirplane* много членов, то написание кода, передающего вызовы от клиента в MyAirplane, будет утомительным. По счастью, это не вопрос поддержки, так как после обнародования интерфейса он не изменяется. Агрегирование дает некоторую поблажку ленивому программисту, который не хочет реализовывать код для передачи вызовов внутренним объектам. Однако при использовании агрегирования к интерфейсу нельзя добавить свой код. Перейдем теперь к агрегированию.

Реализация агрегирования

Рассмотрим, как работает агрегирование. Клиент запрашивает у внешнего компонента интерфейс *IY*. Вместо того, чтобы реализовывать *IY*, внешний компонент запрашивает у внутреннего его интерфейс *IY*, указатель на который и возвращается клиенту. Когда клиент использует интерфейс *IY*, он напрямую вызывает функции-члены *IY*, реализованные внутренним компонентом. В работе клиента с этим интерфейсом

внешний компонент не участвует, полный контроль над интерфейсом *IY* принадлежит внутреннему компоненту.

Хотя поначалу агрегирование кажется простым, как мы увидим далее, правильная реализация интерфейса *IUnknown* внутреннего компонента представляет некоторые трудности. Магия агрегирования заключена в реализации *QueryInterface*. Давайте реализуем *QueryInterface* для внешнего компонента так, чтобы тот возвращал указатель на объект внутреннего.

C++ и агрегирование

В C++ нет средства, эквивалентного агрегированию. Агрегирование — это динамический вид наследования, тогда как наследование C++ всегда статично. Наилучший способ моделирования агрегирования в C++ — это переопределениеоперации разыменования (*operator* ->). Эта техника будет рассматриваться при реализации smart-указателей в следующей главе. Переопределение *operator* -> связано с большими ограничениями, чем агрегирование COM. Вы можете передавать вызовы только одному классу, тогда как в COM можно агрегировать сколько угодно интерфейсов.

Магия *QueryInterface*

Вот объявление внешнего компонента, который реализует интерфейс *IX* и предоставляет интерфейс *IY* посредством агрегирования.

```
class CA : public IX
{
public:
  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
  // Интерфейс IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }
  // Конструктор
  CA();
  // Деструктор
  ~CA();
  // Функция инициализации, вызываемая фабрикой класса для
  // создания включаемого компонента
  HRESULT Init();
private:
  // Счетчик ссылок
  long m_cRef;
  // Указатель на IUnknown внутреннего компонента
  IUnknown* m_pUnknownInner;
};
```

Обратите внимание, что по внешнему виду объявленного компонента нельзя сказать, что он поддерживает интерфейс *IY*: он не наследует *IY* и не реализует какие-либо его члены. Этот внешний компонент использует реализацию *IY* внутреннего компонента. Основные действия внешнего компонента происходят внутри его функции *QueryInterface*, которая возвращает указатель на интерфейс внутреннего объекта. В приведенном ниже фрагменте кода переменная-член *m_pUnknownInner* содержит адрес *IUnknown* внутреннего компонента.

```
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
```

```
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid = IID_IX)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid = IID_IY)
  {
    return m_pUnknownInner->QueryInterface(iid, ppv);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}
```

В этом примере *QueryInterface* внешнего компонента просто вызывает *QueryInterface* внутреннего. Все очень хорошо и просто, но если бы это еще правильно работало! Проблема заключается не в приведенном коде — она в интерфейсе *IUnknown* внутреннего компонента. Агрегированный внутренний компонент должен обрабатывать вызовы функций-членов *QueryInterface* особым образом. Как мы увидим, здесь фактически необходимы две реализации *IUnknown*.

Теперь давайте рассмотрим, почему обычная реализация *IUnknown* для внутреннего компонента вызывает проблемы. Затем познакомимся с тем, как для их устранения реализуются два интерфейса *IUnknown*. Я покажу также, как модифицировать создание внутреннего компонента, чтобы оба компонента получили нужные им указатели. Наконец, мы посмотрим, как внешнему компоненту передаются указатели на интерфейсы внутреннего (это более сложный процесс, чем Вам может показаться).

Неверный *IUnknown*

Задача агрегирования — убедить клиента, что интерфейс, реализованный внутренним компонентом, реализован внешним. Вы должны напрямую передать клиенту указатель на интерфейс внутреннего компонента и внушить ему, что этот указатель принадлежит внешнему компоненту. Если клиенту передать указатель интерфейса, реализованного внутренним компонентом как обычно, то компонент будет представлен клиенту в расщепленном виде. Интерфейс внутреннего компонента использует реализацию *QueryInterface* внутреннего компонента, тогда как у внешнего компонента имеется своя собственная *QueryInterface*. Когда клиент прямо запрашивает интерфейсы внутреннего компонента, у него создается иное представление о возможностях компонента, чем если он запрашивает интерфейс у внешнего компонента (рис. 9.4).

Сказанное можно пояснить примером. Предположим, Вы агрегировали компонент. Внешний компонент поддерживает интерфейсы *IX* и *IY*. Он реализует *IX* и агрегирует *IY*. Внутренний компонент реализует интерфейсы *IY* и *IZ*. Создав внешний компонент, мы получаем указатель на его интерфейс *IUnknown*. С помощью этого интерфейса можно успешно запросить интерфейс *IX* или *IY*, но запрос *IZ* будет всегда возвращать E_NOINTERFACE. Если мы запросим указатель на *IY*, то получим указатель на интерфейс внутреннего компонента. Если запросить *IZ* через этот указатель на *IY*, то запрос будет успешным. Так получается из-за того, что функции интерфейса *IUnknown* для интерфейса *IY* реализованы внутренним компонентом. Точно так же, запрос у интерфейса *IY* интерфейса *IX* потерпит неудачу, поскольку внутренний компонент не поддерживает *IX*.

Такая ситуация нарушает фундаментальное правило реализации *QueryInterface*: если Вы можете попасть в определенное место откуда-нибудь, то туда можно попасть откуда угодно.

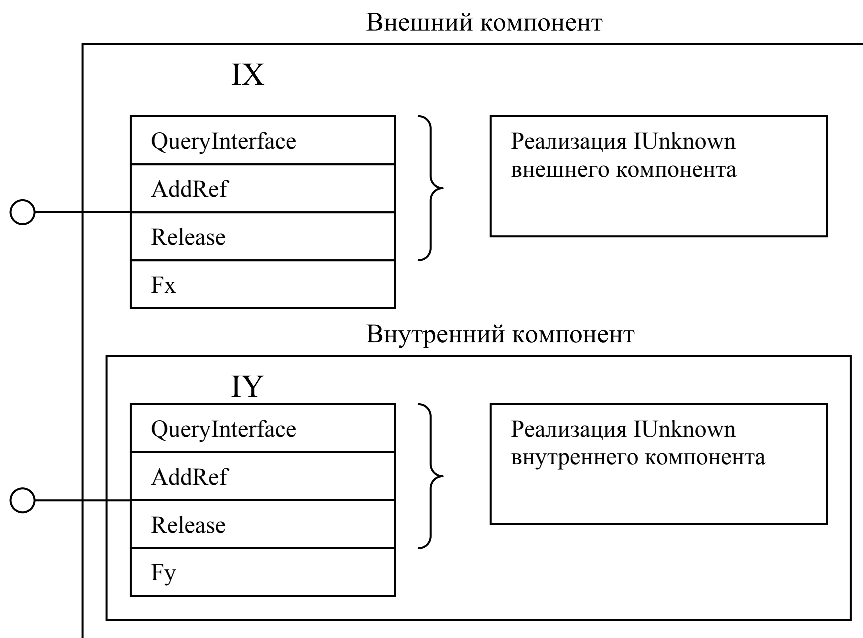


Рис. 9.4 *Разные компоненты имеют разные реализации IUnknown*

Виноват здесь интерфейс *IUnknown* внутреннего компонента. Клиент видит два разных *IUnknown*, внутренний и внешний. Это сбивает его с толку — каждый из *IUnknown* реализует *QueryInterface* по-своему, и каждая *QueryInterface* поддерживает разные наборы интерфейсов. Клиент должен быть абсолютно независим от реализации компонента-агрегата. Он не должен знать, что внешний компонент агрегирует внутренний, и никогда не должен видеть *IUnknown* внутреннего компонента. Как было сказано в гл. 4, два интерфейса реализованы одним и тем же компонентом тогда и только тогда, когда оба они возвращают один и тот же указатель в ответ на запрос указателя на *IUnknown*.

Следовательно, необходимо дать клиенту единственный *IUnknown* и скрыть от него *IUnknown* внутреннего компонента. Интерфейсы внутреннего компонента должны использовать интерфейс *IUnknown*, реализованный внешним компонентом. *IUnknown* внешнего компонента называют *внешним Iunknown (outer unknown)*, или *управляющим IUnknown (controlling unknown)*.

Интерфейсы *IUnknown* для агрегирования

Самый простой для внутреннего компонента способ использовать внешний *IUnknown* — передавать ему вызовы своего *IUnknown*. Для этого внутренний компонент должен знать, что он агрегируется, и должен иметь указатель на внешний *IUnknown*.

Внешний *IUnknown*

Из гл. 8 Вы помните, что функциям *CoCreateInstance* и *IClassFactory::CreateInstance* передается указатель на *IUnknown*, который мы не использовали:

```
HRESULT __stdcall CoCreateInstance(
        const CLSID& clsid,
        IUnknown* pUnknownOuter, // Внешний компонент
        DWORD dwClsContext, // Контекст сервера
        const IID& iid,
        void** ppv
);

HRESULT __stdcall CreateInstance(
        IUnknown* pUnknownOuter,
        const IID& iid,
        void** ppv
);
```

Внешний компонент передает указатель на свой интерфейс *IUnknown* внутреннему компоненту с помощью параметра *pUnknownOuter*. Если указатель на внешний *IUnknown* не равен NULL, то компонент агрегируется.

Используя указатель на *IUnknown*, переданный *CreateInstance*, компонент узнает, агрегируется ли он и кто его агрегирует. Если компонент не агрегируется, он использует собственную реализацию *IUnknown*. В противном случае он должен делегировать вызовы внешнему *IUnknown*.

Делегирующий и неделегирующий *IUnknown*

Для поддержки агрегирования внутренний компонент фактически реализует два интерфейса *IUnknown*.

Неделегирующий (nondelegating) IUnknown реализует *IUnknown* внутреннего компонента обычным образом.

Делегирующий (delegating) IUnknown передает вызовы методов *IUnknown* либо внешнему *IUnknown*, либо неделегирующему *IUnknown*. Если внутренний компонент агрегируется, делегирующий *IUnknown* передает вызовы внешнему *IUnknown*, реализованному внешним компонентом. Клиенты агрегата вызывают делегирующий *IUnknown*, тогда как внешний компонент работает с внутренним через неделегирующий. Эту ситуацию поясняют рисунки: вариант без агрегирования показан на рис. 9.5, вариант с агрегированием — на рис. 9.6.

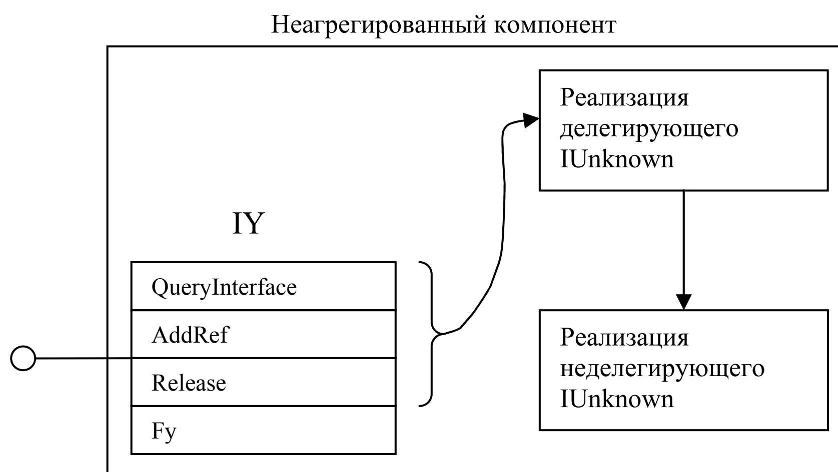


Рис. 9.5 *Если компонент не агрегируется, его делегирующий IUnknown передает вызовы неделегирующему IUnknown*

На рис. 9.6 представлен компонент, агрегирующий *IY*. В этом случае делегирующий *IUnknown* вызывает *IUnknown*, реализованный внешним компонентом. Внешний компонент вызывает неделегирующий *IUnknown* для управления временем существования внутреннего компонента. Таким образом, когда некоторый компонент вызывает метод *IUnknown* через указатель на интерфейс *IY*, он вызывает делегирующий *IUnknown*, который перенаправляет вызов внешнему *IUnknown*. В результате внутренний компонент использует реализацию *IUnknown* внешнего компонента.

Теперь все, что мы должны сделать, — это реализовать делегирующий и неделегирующий *IUnknown*.

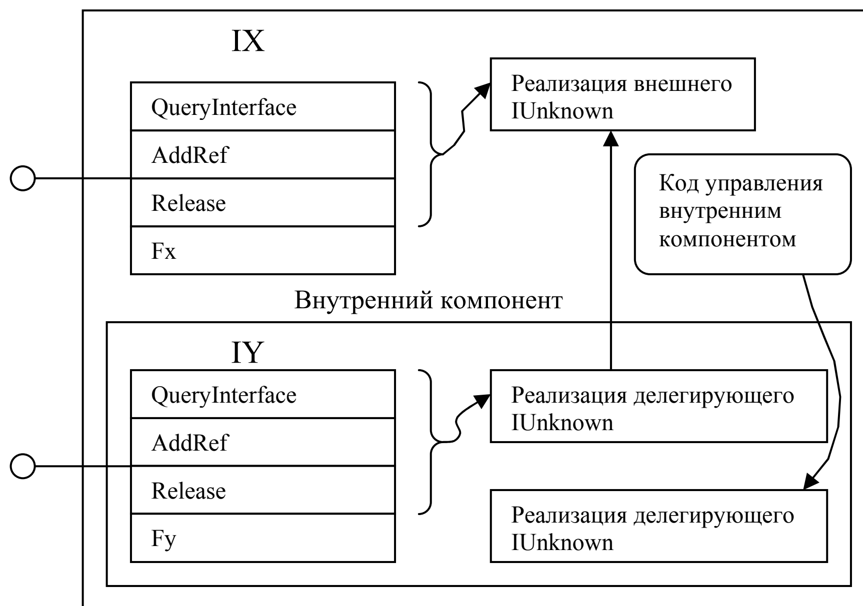


Рис. 9.6 *Если компонент агрегирован, его делегирующий Iunknown передает вызовы внешнему IUnknown*

Реализация делегирующего и неделегирующего *IUnknown*

Нам необходимо иметь для компонента две разные реализации *IUnknown*. Но С++ не позволяет реализовать интерфейс дважды в одном классе. Следовательно, мы изменим имя одного из *IUnknown*, чтобы избежать конфликта имен. Я выбрал имя *InondelegatingUnknown*. Вы можете выбрать то, которое Вам больше нравится.
Вспомните, что для СОМ имена интерфейсов не имеют значения; СОМ интересуется только структурой *vtbl*. *InondelegatingUnknown* объявлен так же, как и *IUnknown*, за исключением того, что названия этих функций-членов имеют префикс «Nondelegating».

```
struct InondelegatingUnknown
{
  virtual HRESULT __stdcall NondelegatingQueryInterface(const IID&, void**) =
0;
  virtual ULONG __stdcall NondelegatingAddRef() = 0;
  virtual ULONG __stdcall NondelegatingRelease() = 0;
};
```

Методы *NondelegatingAddRef* и *NondelegatingRelease* интерфейса *InondelegatingUnknown* реализованы в точности так же, как ранее были реализованы *AddRef* и *Release* для *IUnknown*. Однако в *NondelegatingQueryInterface* есть небольшое, но очень важное изменение.

```
HRESULT __stdcall CB::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    *ppv = static_cast<INondelegatingUnknown*>(this);
  }
  else if (iid = IID_IY)
  {
    *ppv = static_cast<IY*>(this);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}
```

Обратите внимание на приведение типа указателя *this* внутреннего компонента к *INondelegatingUnknown*. Это приведение очень важно. Преобразуя *this* к *INondelegatingUnknown*, мы гарантируем, что будет возвращен неделегирующий *IUnknown*. Неделегирующий *IUnknown* всегда возвращает указатель на себя, если у него запрашивается *IID_IUnknown*. Без этого приведения типа вместо неделегирующего *IUnknown* возвращался бы делегирующий. Когда компонент агрегируется, делегирующий *IUnknown* передает все вызовы *QueryInterface*, *Release* и *AddRef* внешнему объекту.

Клиенты агрегируемого компонента никогда не получают указатели на неделегирующий *IUnknown* внутреннего компонента. Всякий раз, когда клиент запрашивает указатель на *IUnknown*, он получает *IUnknown* внешнего компонента. Указатель на неделегирующий *IUnknown* внутреннего компонента передается только внешнему компоненту. Теперь рассмотрим, как реализовать делегирующий *IUnknown*.

Реализация делегирующего *IUnknown*

К счастью, реализация делегирующего *IUnknown* проста — она передает вызовы либо внешнему, либо неделегирующему *IUnknown*. Далее приводится объявление компонента, поддерживающего агрегирование. Компонент содержит указатель *m_pUnknownOuter*. Если компонент агрегирован, указатель ссылается на внешний *IUnknown*. Если компонент не агрегирован, этот указатель ссылается на неделегирующий *IUnknown*. Всякий раз при обращении к делегирующему *IUnknown* вызов переадресуется интерфейсу, на который указывает *m_pUnknownOuter*. Делегирующий *IUnknown* реализован функциями, подставляемыми в строку (inline):

```
class CB : public IY, INondelegatingUnknown
{
public:
  // Делегирующий IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv)
  {
    // Делегировать QueryInterface
    return m_pUnknownOuter->QueryInterface(iid, ppv);
  }
  virtual ULONG __stdcall AddRef()
  {
    // Делегировать AddRef
    return m_pUnknownOuter->AddRef();
  }
  virtual ULONG __stdcall Release()
```

```
  {
    // Делегировать Release
    return m_pUnknownOuter->Release();
  }
  // Неделегирующий IUnknown
  virtual HRESULT __stdcall
  NondelegatingQueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall NondelegatingAddRef();
  virtual ULONG __stdcall NondelegatingRelease();
  // Интерфейс IY
  virtual void Fy() { cout << "Fy" << endl; }
  // Конструктор
  CB(IUnknown* pUnknownOuter);
  // Деструктор
  ~CB();
private:
  long m_cRef;
  IUnknown* m_pUnknownOuter;
};
```

Создание внутреннего компонента

Теперь, когда мы знаем, как реализовать внутренний компонент, обсудим, как он создается внешним компонентом. Чтобы пройти весь процесс создания, от начала до конца, рассмотрим код трех функций: функция *Init* внешнего компонента начинает процесс; затем вступает функция фабрики класса *CreateInstance* и конструктор внутреннего компонента.

Функция *Init* внешнего компонента

Первое, что делает внешний компонент при агрегировании — создает внутренний компонент. Основное различие между включением и агрегированием состоит в том, что во втором случае внешний компонент передает внутреннему внешний *IUnknown*. Приведенный ниже фрагмент кода показывает, как внешний компонент создает внутренний. Обратите внимание, что второй параметр *CoCreateInstance* — это указатель на интерфейс *Iunknown* внешнего компонента.

Кроме того, отметьте себе, что пятый параметр запрашивает у внутреннего компонента указатель на *IUnknown*. Фабрика класса будет возвращать указатель на неделегирующий *IUnknown* внутреннего компонента. Как мы уже видели, это указатель необходим внешнему компоненту для передачи вызовов *QueryInstance* внутреннему компоненту. Здесь внешний компонент обязан запрашивать указатель на *IUnknown*; в противном случае он никогда не сможет получить его. Внешний компонент должен сохранить неделегирующий *IUnknown* внутреннего компонента для последующего использования.

В данном примере нет необходимости явно приводить указатель *this* к указателю на *IUnknown*, так как *CA* наследует только *IX*, и, следовательно, неявное преобразование не будет неоднозначным.

```
HRESULT CA::Init()
{
  IUnknown* pUnknownOuter = this;
  HRESULT hr = CoCreateInstance(CLSID_Component2,
  pUnknownOuter,
  CLSCTX_INPROC_SERVER,
  IID_IUnknown,
  (void**)&m_pUnknownOuter);
  if (FAILED(hr))
```

```
  {
    return E_FAIL;
  }
  return S_OK;
}
```

Реализация *IClassFactory::CreateInstance* внешнего компонента вызывает *CA::Init.* В других отношениях реализация *IClassFactory* внешнего компонента остается неизменной. Фабрика же класса внутреннего компонента подверглась некоторым изменениям, которые мы теперь и рассмотрим.

Функция *IClassFactory::CreateInstance* внутреннего компонента

Реализация *IClassFactory::CreateInstance* внутреннего компонента изменена, чтобы использовать *InondelegatingUnknown* вместо *IUnknown*. Код этой функции приведен ниже, отличия от предыдущих вариантов *CreateInstance* не возвращает автоматически ошибку, если *pUnknownOuter* не равен NULL (т.е. когда внешний компонент желает агрегировать внутренний). Однако *CreateInstance* обязана возвратить ошибку, если при этом *iid* отличен от *IID_IUnknown*. Когда компонент создается как внутренний в агрегате, он может возвратить только интерфейс *IUnknown*, иначе внешний компонент никогда не получил бы неделегирующего *IUnknown* (поскольку вызовы *QueryInterface* будут делегированы внешнему *IUnknown*).

```
HRESULT __stdcall Cfactory::CreateInstance(IUnknown* pUnknownOuter,
const IID& iid,
void** ppv)
{
  // При агрегировании iid должен быть IID_IUnknown
  if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
  {
    return CLASS_E_NOAGGREGATION;
  }
   // Создать компонент
  CB* pB = new CB(pUnknownOuter);
  if (pB == NULL)
  {
    return E_OUTOFMEMORY;
  }
  // Получить запрошенный интерфейс
  HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv);
  PB->NondelegatingRelease();
  return hr;
}
```

Для получения запрашиваемого интерфейса вновь созданного внутреннего компонента показанная выше функция *CreateInstance* вызывает не *QueryInterface*, а *NondelegatingQueryInterface*. Если внутренний компонент агрегируется, вызовы *QueryInterface* он будет делегировать внешнему *IUnknown*. Фабрика класса должна возвратить указатель на неделегирующий *QueryInterface*, поэтому он вызывает *NondelegatingQueryInterface*.

Конструктор внутреннего компонента

В приведенном выше коде *CreateInstance* указатель на внешний *IUnknown* передается конструктору внутреннего компонента. Конструктор инициализирует *m_pUnknownOuter*, которая используется делегирующим *Iunknown* для передачи вызовов либо неделегирующему, либо внешнему *IUnknown*. Если компонент не агрегируется

(*pUnknownOuter* есть NULL), конструктор помещает в *m_pUnknownOuter* указатель на неделегирующий *IUnknown*. Это показано в приведенном ниже фрагменте:

```
CB::CB(IUnknown* pUnknownOuter) : m_cRef(1)
{
  ::InterlockedIncrement)&g_cComponents;
  if (pUnknownOuter == NULL)
  {
    // Не агрегируется: использовать неделегирующий IUnknown
    m_pUnknownOuter = reinterpret_cast<IUnknown*>(
    static_cast<INondelegatingUnknown*>(this)
    );
  }
  else
  {
    // Агрегируется: использовать внешний IUnknown
    m_pUnknownOuter = pUnknownOuter;
  }
}
```

Указатели внешнего компонента на интерфейсы внутреннего компонента

Когда я реализовывал *CA::Init* при создании внутреннего компонента, я запрашивал интерфейс *IUnknown*, а не *IY*. Однако наш компонент в действительности агрегирует *IY*. Поэтому неплохо было бы в самом начале проверить, поддерживается ли интерфейс *IY* внутренний компонент. Но, как указывалось выше, при агрегировании компонента внешний компонент может запрашивать только интерфейс *IUnknown*.

```
Cfactory::CreateInstance
```

возвращает CLASS_E_NOAGGREGATION, если ей передано что-либо, отличное от *IID_IUnknown*. Следовательно, нам необходимо запросить у внутреннего компонента интерфейс *IY* после его (компонента) создания.

Но здесь необходимо быть аккуратным и не запутаться. Когда Вы вызываете *QueryInterface*, чтобы получить по *m_pUnknownInner* указатель на интерфейс *IID_IY*, эта функция, как примерный школьник, вызывает для возвращаемого указателя *AddRef*. Поскольку внутренний компонент агрегирован, он делегирует вызов *AddRef* внешнему *IUnknown*. В результате увеличивается счетчик ссылок внешнего компонента, а не внутреннего. Я хочу еще раз это подчеркнуть. Когда внешний компонент запрашивает интерфейс через указатель на неделегирующий *IUnknown* или какой-либо еще интерфейс внутреннего компонента, счетчик ссылок внешнего компонента увеличивается. Это именно то, что требуется, когда интерфейс через указатель на интерфейс внутреннего компонента запрашивает клиент. Но в данном случае указатель на интерфейс *IY* запрашивает внешний компонент, и счетчик ссылок для этого указателя является счетчиком ссылоквнешнего компонента. Таким образом, внешний компонент удерживает одну ссылку сам на себя! Если допустить такое, счетчик ссылок внешнего компонента никогда не станет нулем, и компонент никогда не будет удален из памяти.

Так как время существования указателя на *IY*, принадлежащего внешнему компоненту, вложено во время существования самого внешнего компонента, нам нет необходимости увеличивать счетчик ссылок. Но не вызывайте для уменьшения счетчика ссылок *Release* для *IY* — мы обязаны работать с интерфейсом *IY* так, как если бы у него был отдельный счетчик ссылок. (В нашей реализации компонента неважно, для какого указателя вызывать *Release*. Но в других случаях это может быть не так.) Освобождение интерфейса *IY* может освободить используемые им ресурсы. Следовательно, общее

правило состоит в том, чтобы вызывать *Release*, используя указатель, переданный в *CoCreateInstance*. Версия *CA::Init*, запрашивающая интерфейс *IY*, приведена ниже:

```
HRESULT __stdcall CA::Init()
{
  // Получить указатель на внешний IUnknown
  IUnknown* pUnknownOuter = this;
  // Создать внутренний компонент
  HRESULT hr = CoCreateInstance(CLSID_Component2,
  PUnknownOuter, // IUnknown внешнего компонента
  CLSCTX_INPROC_SERVER,
  IID_IUnknown, // При агрегировании только IUnknown
  (void**)&m_pUnknownInner);
  if (FAILED(hr))
  {
    // Ошибка при создании компонента
    return E_FAIL;
  }
  // Этот вызов увеличит счетчик ссылок внешнего компонента
  // Получить интерфейс IY внутреннего компонента
  hr = m_pUnknownInner->QueryInterface(IID_IY, (void**)&m_pIY);
  if (FAILED(hr))
  {
     // Внутренний компонент не поддерживает интерфейс IY
     m_pUnknownInner->Release();
     return E_FAIL;
  }
  // Нам нужно уменьшить счетчик ссылок на внешний компонент,
  // увеличенный предыдущим вызовом
  pUnknownOuter->Release();
  return S_OK;
}
```

При реализации *QueryInterface* у нас есть выбор — либо возвращать *m_pIY*, либо вызывать *QueryInterface* внутреннего компонента. Поэтому можно использовать либо

```
else if (iid == IID_IY)
{
   return m_pUnknownOuter->QueryInterface(iid, ppv);
}
как мы это делали, либо
else if (iid == IID_IY)
{
*ppv = m_pIY;
}
```

Итак, мы уже создали внутренний компонент, запросили у него интерфейс, скорректировали счетчик ссылок и вернули интерфейс клиенту. Чего мы еще не сделали, так это не освободили интерфейс в деструкторе внешнего компонента. Мы не можем просто вызвать *m_pIY->Release*, так как у нас нет для него подсчитанной ссылки. Мы убрали ее в функции *Init* внешнего компонента после того, как получили указатель на *IY*. Теперь необходимо повторить процедуру в обратном порядке, восстановив счетчик ссылок и вызвав *Release* для указателя на *IY*.

Однако здесь следует быть осторожным, так как в противном случае этот последний вызов *Release* снова сделает счетчик ссылок внешнего компонента нулевым, и тот попытается удалить себя. Таким образом, процесс освобождения нашего указателя на интерфейс внутреннего компонента состоит из трех этапов. Во-первых, необходимо гарантировать, что наш компонент не попытается снова удалить себя. Во-вторых,

необходимо вызвать *AddRef* для внешнего компонента, так как любой вызов *Release* для внутреннего компонента вызовет *Release* для внешнего. И, наконец, мы можем освободить указатель на *IY*, хранящийся во внешнем компоненте. Соответствующий код приведен ниже:

```
// 1. Увеличить счетчик ссылок во избежание
// рекурсивного вызова деструктора
  m_cRef = 1;

// 2. AddRef для внешнего IUnknown
  IUnknown* pUnknownOuter = this;
  pUnknownOuter->AddRef();

// 3. Освободить интерфейс
  m_pIY->Release();
```

Давайте мысленно проследим работу этого кода внешнего компонента. Первое, что мы делаем, — устанавливаем счетчик ссылок в 1. Далее мы увеличиваем его до двух. Затем вызываем *Release* для интерфейса *IY*. Внутренний компонент будет делегировать этот вызов внешнему. Последний уменьшит счетчик ссылок с 2 до 1. Если бы мы не установили ранее счетчик ссылок в 1, то компонент попытался бы во второй раз удалить себя. В нашей реализации, когда внутренний компонент агрегируется, его функция *Release* просто делегируется внешнему *IUnknown*. Однако внешний компонент должен работать с внутренним так, как если бы тот вел отдельные счетчики ссылок для каждого интерфейса, поскольку другие реализации внутреннего компонента могут делать больше, чем просто делегировать *Release* внешнему компоненту. Внутренний компонент может также освобождать некоторые ресурсы или выполнять другие операции.

Большая часть этого кода может показаться избыточной и ненужной. Но если внешний компонент сам агрегирован другим компонентом, выполнение описанных выше шагов становится критически важным. В примере из гл. 10 показан компонент, который агрегирует компонент, агрегирующий другой компонент. «Вот и все, что необходимо сделать для реализации агрегирования», — сказал он, улыбаясь. В действительности, после того, как Вы написали корректный код агрегирования, он работает отлично, и о нем можно забыть. Однако после первой попытки его написать многие начинают называть его не «aggregation», а «aggravation»*.

Законченный пример

Реализуем компонент, который агрегирует некий интерфейс. В данном примере Компонент 1 поддерживает два интерфейса, так же как и в примере с включением. Однако здесь он реализует только *IX*. Он не будет ни реализовывать *IY*, ни передавать его вызовы реализации этого интерфейса Компонентом 2. Вместо этого, когда клиент запрашивает у Компонента 1 интерфейс *IY*, Компонент 1 возвращает указатель на интерфейс *IY*, реализуемый внутренним Компонентом 2. В листинге 8-3 представлен внешний компонент, а в листинге 9.4 — внутренний. Клиент остался практически неизменным; ему совершенно неважно, используем ли мы агрегирование или включение.

AGGRGATE\CMPNT1

```
//
// Cmpnt1.cpp - Компонент 1
//
// Интересные части кода выделены полужирным шрифтом
//

#include <iostream.h>
#include <objbase.h>

#include "Iface.h"
#include "Registry.h"

// Функция Trace
void trace(const char* msg) { cout << "Компонент 1:\t" << msg << endl; }

///////////////////////////////////////////////////////////
//
// Глобальные переменные
//

// Статические переменные
static HMODULE g_hModule = NULL; // Дескриптор модуля DLL
static long g_cComponents = 0; // Счетчик активных компонентов
static long g_cServerLocks = 0; // Число блокировок

// Дружественное имя компонента
const char g_szFriendlyName[] = "Основы COM, Глава 8 Пример 2, Компонент 1";

// Не зависящий от версии ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap09.Ex2.Cmpnt1";

// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt1.1";

///////////////////////////////////////////////////////////
//
// Компонент A
//
class CA : public IX //, public IY
{
public:

  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // Интерфейс IX
  virtual void __stdcall Fx() { cout << "Fx" << endl; }

  /* Компонент 1 агрегирует интерфейс IY, а не реализует его
  // Интерфейс IY
  virtual void __stdcall Fy() { m_pIY->Fy(); }
  */

  // Конструктор
  CA();

  // Деструктор
  ~CA();
```

```
  // Функция инициализации, вызываемая фабрикой класса для
  // создания агрегируемого компонента
  HRESULT __stdcall Init();

private:
  // Счетчик ссылок
  long m_cRef;

  // Указатель на интерфейс IY агрегированного компонента
  // (Нам необязательно сохранять указатель на IY. Однако мы
  // можем использовать его в QueryInterface)
  IY* m_pIY;
  // Указатель на IUnknown внутреннего компонента
  IUnknown* m_pUnknownInner;
};

//
// Конструктор
//
CA::CA() : m_cRef(1), m_pUnknownInner(NULL)
{
  ::InterlockedIncrement(&g_cComponents);
}

//
// Деструктор
//
CA::~CA()
{
  ::InterlockedDecrement(&g_cComponents);
  trace("Самоликвидация");

  // Предотвращение рекурсивного вызова деструктора следующей
  // ниже пары AddRef/Release
  m_cRef = 1;

  // Учесть pUnknownOuter->Release в методе Init
  IUnknown* pUnknownOuter = this;
  pUnknownOuter->AddRef();

  // Правильное освобождение указателя; возможен поинтерфейсный
  // подсчет ссылок
  m_pIY->Release();
  // Освободить внутренний компонент
  if (m_pUnknownInner != NULL)
  {
    m_pUnknownInner->Release();
  }
}

// Инициализировать компонент путем создания внутреннего компонента
HRESULT __stdcall CA::Init()
{
  // Получить указатель на внешний IUnknown
  // Поскольку этот компонент агрегируется, внешний IUnknown -
  // это то же самое, что и указатель this
  IUnknown* pUnknownOuter = this;
  trace("Создать внутренний компонент");
  HRESULT hr = ::CoCreateInstance(CLSID_Component2,
                                  pUnknownOuter, // IUnknown внешнего компонента
                                  CLSCTX_INPROC_SERVER,
                                  IID_IUnknown, // При агрегировании - IUnknown
                                  (void**)&m_pUnknownInner);
```

```
  if (FAILED(hr))
  {
    trace("Не могу создать внутренний компонент");
    return E_FAIL;
  }

  // Следующий вызов будет увеличивать счетчик ссылок внешнего компонента
  trace("Получить интерфейс IY внутреннего компонента");
  hr = m_pUnknownInner->QueryInterface(IID_IY, (void**)&m_pIY);
  if (FAILED(hr))
  {
    trace("Внутренний компонент не поддерживает интерфейс IY");
    m_pUnknownInner->Release();
    return E_FAIL;
  }

  // Необходимо уменьшить счетчик ссылок внешнего компонента, увеличенный
  // предыдущим вызовом. Для этого вызываем Release для указателя,
  // переданного CoCreateInstance.
  pUnknownOuter->Release();
  return S_OK;
}

//
// Реализация IUnknown
//
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    *ppv = static_cast<IUnknown*>(this);
  }
  else if (iid == IID_IX)
  {
    *ppv = static_cast<IX*>(this);
  }
  else if (iid == IID_IY)
  {
    trace("Вернуть интерфейс IY внутреннего компонента");
    #if 1
    // Этот интерфейс можно запросить...
    return m_pUnknownInner->QueryInterface(iid,ppv);
    #else
    // либо можно вернуть сохраненный указатель
    *ppv = m_pIY;
    // Проходим дальше, чтобы была вызвана AddRef
    #endif
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CA::AddRef()
{
  return ::InterlockedIncrement(&m_cRef);
}
```

```
ULONG __stdcall CA::Release()
{
  if (::InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

///////////////////////////////////////////////////////////
//
// Фабрика класса
//
class CFactory : public IClassFactory
{
public:

  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // Интерфейс IClassFactory
  virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
  const IID& iid,
  void** ppv);
  virtual HRESULT __stdcall LockServer(BOOL bLock);

  // Конструктор
  CFactory() : m_cRef(1) {}

  // Деструктор
  ~CFactory() {}

private:
  long m_cRef;
};

//
// Реализация IUnknown для фабрики класса
//
HRESULT __stdcall CFactory::QueryInterface(REFIID iid, void** ppv)
{
  IUnknown* pI;
  if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
  {
    pI = static_cast<IClassFactory*>(this);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  pI->AddRef();
  *ppv = pI;
  return S_OK;
}

ULONG __stdcall CFactory::AddRef()
{
  return ::InterlockedIncrement(&m_cRef);
}
```

```
ULONG __stdcall CFactory::Release()
{
  if (::InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

//
// Реализация IClassFactory
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
  // Агрегирование не поддерживается
  if (pUnknownOuter != NULL)
  {
    return CLASS_E_NOAGGREGATION;
  }

  // Создать компонент
  CA* pA = new CA;
  if (pA == NULL)
  {
    return E_OUTOFMEMORY;
  }

  // Инициализировать компонент
  HRESULT hr = pA->Init();
  if (FAILED(hr))
  {
    // Ошибка инициализации. Удалить компонент.
    pA->Release();
    return hr;
  }

  // Получить запрошенный интерфейс
  hr = pA->QueryInterface(iid, ppv);
  pA->Release();
  return hr;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
  if (bLock)
  {
    ::InterlockedIncrement(&g_cServerLocks);
  }
  else
  {
    ::InterlockedDecrement(&g_cServerLocks);
  }
  return S_OK;
}


///////////////////////////////////////////////////////////
//
// Экспортируемые функции
//
```

```
STDAPI DllCanUnloadNow()
{
  if ((g_cComponents == 0) && (g_cServerLocks == 0))
  {
    return S_OK;
  }
  else
  {
    return S_FALSE;
  }
}

//
// Получение фабрики класса
//

STDAPI DllGetClassObject(const CLSID& clsid,
const IID& iid,
void** ppv)
{
  // Можем ли мы создать такой компонент?
  if (clsid != CLSID_Component1)
  {
    return CLASS_E_CLASSNOTAVAILABLE;
  }

  // Создать фабрику класса
  CFactory* pFactory = new CFactory; // В конструкторе нет Addref
  if (pFactory == NULL)
  {
    return E_OUTOFMEMORY;
  }

  // Получить запрошенный интерфейс
  HRESULT hr = pFactory->QueryInterface(iid, ppv);
  pFactory->Release();
  return hr;
}

//
// Регистрация сервера
//
STDAPI DllRegisterServer()
{
  return RegisterServer(g_hModule,
  CLSID_Component1,
  g_szFriendlyName,
  g_szVerIndProgID,
  g_szProgID);
}

STDAPI DllUnregisterServer()
{
  return UnregisterServer(CLSID_Component1,
  g_szVerIndProgID,
  g_szProgID);
}



///////////////////////////////////////////////////////////
//
// Информация о модуле DLL
//
```

```
BOOL APIENTRY DllMain(HANDLE hModule,
DWORD dwReason,
void* lpReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    g_hModule = hModule;
  }
  return TRUE;
}
```

Листинг 9.3 *Реализация внешнего (агрегирующего) компонента*

AGGRGATE\CMPNT2

```
//
// Cmpnt2.cpp - Компонент 2
// Помните, что изменения в фабрике класса отмечены полужирным шрифтом
//

#include <iostream.h>
#include <objbase.h>

#include "Iface.h"
#include "Registry.h"

void trace(const char* msg) { cout << "Компонент 2:\t" << msg << endl; }

///////////////////////////////////////////////////////////
//
// Глобальные переменные
//

// Статические переменные
static HMODULE g_hModule = NULL; // Дескриптор модуля DLL
static long g_cComponents = 0; // Счетчик активных компонентов
static long g_cServerLocks = 0; // Количество блокировок

// Дружественное имя компонента
const char g_szFriendlyName[]
= "Основы COM, Глава 8 Пример 2, Компонент 2";

// Независящий от версии ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2";

// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2.1";

///////////////////////////////////////////////////////////
//
// Неделегирующий интерфейс IUnknown
//

struct INondelegatingUnknown
{
  virtual HRESULT __stdcall
  NondelegatingQueryInterface(const IID&, void**) = 0;
  virtual ULONG __stdcall NondelegatingAddRef() = 0;
  virtual ULONG __stdcall NondelegatingRelease() = 0;
};

///////////////////////////////////////////////////////////
//
// Компонент B
//
```

```
class CB : public IY, public INondelegatingUnknown
{
public:
  // Делегирующий IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv)
  {
    trace("Делегирующий QueryInterface");
    return m_pUnknownOuter->QueryInterface(iid, ppv);
  }
  virtual ULONG __stdcall AddRef()
  {
    trace("Делегировать AddRef");
    return m_pUnknownOuter->AddRef();
  }
  virtual ULONG __stdcall Release()
  {
    trace("Делегировать Release");
    return m_pUnknownOuter->Release();
  }

  // Неделегирующий IUnknown
  virtual HRESULT __stdcall
  NondelegatingQueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall NondelegatingAddRef();
  virtual ULONG __stdcall NondelegatingRelease();

  // Интерфейс IY
  virtual void __stdcall Fy() { cout << "Fy" << endl; }

  // Конструктор
  CB(IUnknown* m_pUnknownOuter);

  // Деструктор
  ~CB();
private:
  long m_cRef;
  IUnknown* m_pUnknownOuter;
};

//
// Реализация IUnknown
//
HRESULT __stdcall CB::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    // !!! ПРИВЕДЕНИЕ ТИПА ОЧЕНЬ ВАЖНО !!!
    *ppv = static_cast<INondelegatingUnknown*>(this);
  }
  else if (iid == IID_IY)
  {
    *ppv = static_cast<IY*>(this);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}
```

```
ULONG __stdcall CB::NondelegatingAddRef()
{
  return ::InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CB::NondelegatingRelease()
{
  if (::InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

//
// Конструктор
//

CB::CB(IUnknown* pUnknownOuter) : m_cRef(1)
{
  ::InterlockedIncrement(&g_cComponents);
  if (pUnknownOuter == NULL)
  {
    trace("Не агрегируется; использовать неделегирующий IUnknown");
    m_pUnknownOuter = reinterpret_cast<IUnknown*>
     (static_cast<INondelegatingUnknown*>
     (this));
  }
  else
  {
    trace("Агрегируется; делегировать внешнему IUnknown");
    m_pUnknownOuter = pUnknownOuter;
  }
}

//
// Деструктор
//

CB::~CB()
{
  ::InterlockedDecrement(&g_cComponents);
  trace("Саморазрушение");
}

///////////////////////////////////////////////////////////
//
// Фабрика класса
//
class CFactory : public IClassFactory
{
public:

  // IUnknown
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // Интерфейс IClassFactory
  virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                  const IID& iid,
                                  void** ppv);
  virtual HRESULT __stdcall LockServer(BOOL bLock);
```

```
  // Конструктор
  CFactory() : m_cRef(1) {}

// Деструктор
  ~CFactory() {}
private:
  long m_cRef;
};

//
// Реализация IUnknown для фабрики класса
//

HRESULT __stdcall CFactory::QueryInterface(const IID& iid, void** ppv)
{
  if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
  {
    *ppv = static_cast<IClassFactory*>(this);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
  reinterpret_cast<IUnknown*>(*ppv)->AddRef();
  return S_OK;
}

ULONG __stdcall CFactory::AddRef()
{
  return ::InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CFactory::Release()
{
  if (::InterlockedDecrement(&m_cRef) == 0)
  {
    delete this;
    return 0;
  }
  return m_cRef;
}

//
// Реализация IClassFactory
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
  // При агрегировании iid должен быть IID_IUnknown
  if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
  {
    return CLASS_E_NOAGGREGATION;
  }

  // Создать компонент
  CB* pB = new CB(pUnknownOuter);
  if (pB == NULL)
  {
    return E_OUTOFMEMORY;
  }
```

```
  // Получить запрошенный интерфейс
  HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv);
  pB->NondelegatingRelease();
  return hr;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
  if (bLock)
  {
    ::InterlockedIncrement(&g_cServerLocks);
  }
  else
  {
    ::InterlockedDecrement(&g_cServerLocks);
  }
  return S_OK;
}

///////////////////////////////////////////////////////////
//
// Экспортируемые функции
//
STDAPI DllCanUnloadNow()
{
  if ((g_cComponents == 0) && (g_cServerLocks == 0))
  {
    return S_OK;
  }
  else
  {
    return S_FALSE;
  }
}

//
// Получение фабрики класса
//
STDAPI DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv)
{
  // Можем ли мы создать такой компонент?
  if (clsid != CLSID_Component2)
  {
    return CLASS_E_CLASSNOTAVAILABLE;
  }

// Создать фабрику класса
CFactory* pFactory = new CFactory; // В конструкторе нет AddRef
if (pFactory == NULL)
{
  return E_OUTOFMEMORY;
}

// Получить запрошенный интерфейс
HRESULT hr = pFactory->QueryInterface(iid, ppv);
  pFactory->Release();
  return hr;
}

//
// Регистрация сервера
//
STDAPI DllRegisterServer()
```

```
{
  return RegisterServer(g_hModule,
  CLSID_Component2,
  g_szFriendlyName,
  g_szVerIndProgID,
  g_szProgID);
}

STDAPI DllUnregisterServer()
{
  return UnregisterServer(CLSID_Component2,
  g_szVerIndProgID,
  g_szProgID);
}

///////////////////////////////////////////////////////////
//
// Информация о модуле DLL
//
BOOL APIENTRY DllMain(HANDLE hModule,
DWORD dwReason,
void* lpReserved)
{
  if (dwReason == DLL_PROCESS_ATTACH)
  {
    g_hModule = hModule;
  }
  return TRUE;
}
```

Листинг 9.4 *Реализация внутреннего (агрегируемого) компонента*

Слепое агрегирование

В предыдущем примере внешний компонент агрегирует только один из реализуемых внутренним компонентом интерфейсов. Единственный интерфейс внутреннего компонента, до которого может добраться клиент, это *IY*. Если бы внутренний компонент реализовывал *IZ*, клиент не смог бы получить указатель на *IZ*, так как внешний компонент возвращал бы E_NOINTERFACE.

А что, если внешний компонент хочет агрегировать несколько интерфейсов внутреннего? Внешний компонент легко модифицировать так, чтобы поддерживать еще один интерфейс внутреннего компонента:

```
else is ((iid == IID_IY) || (iid == IID_IZ))
{
  return m_pUnknownInner->QueryInterface(iid, ppv);
}
```

Конечно, для изменения кода внешнего компонента нам потребуется доступ к этому коду и компилятор. Но как быть, если мы хотим, чтобы клиент имел доступ ко всем интерфейсам внутреннего компонента, включая то, что будут добавлены после того, как внешний компонент будет написан, скомпилирован и поставлен заказчику? Все очень просто — удалите условие. Вместо проверки идентификатора интерфейса можно просто слепо передавать его внутреннему компоненту:

```
...
else if (iid == IID_IX)
{
  *ppv = static_cast<IX*>(this);
}
else // Нет условия
{
  return m_pUnknownInner->QueryInterface(iid, ppv);
}
...
```

Данная процедура называется *слепым агрегированием (blind aggregation)*, так как внешний компонент слепо передает идентификаторы интерфейсов внутреннему. При слепом агрегировании внешний компонент не контролирует, какие из интерфейсов внутреннего компонента он предоставляет клиенту. В большинстве случаев лучше не прибегать к слепому агрегированию. Одна из причин этого в том, что внутренний компонент может поддерживать интерфейсы, не совместимые с интерфейсами, которые поддерживаются внешним компонентом.

Например, внешний компонент для сохранения файлом может поддерживать интерфейс *ISlowFile*, тогда как внутренний для этой же цели поддерживает интерфейс *IFastFile*. Предположим, что клиент всегда запрашивает *IFastFile* прежде, чем запрашивать *ISlowFile*. Если внешний компонент слепо агрегирует внутренний, клиент получит указатель на *IFastFile* внутреннего компонента. Внутренний компонент ничего не знает о внешнем и поэтому не будет правильно сохранять информацию, связанную с последним. Проще всего избежать таких конфликтов, не используя слепое агрегирование.
Если полное воздержание кажется излишне сильным условием, для устранения подобных конфликтов можно использовать два менее радикальных способа. Во-первых, при реализации внешнего компонента не реализуйте интерфейсы, которые могут дублировать функциональность интерфейсов внутреннего компонента. Во-вторых, можно создавать внешний компонент и клиент либо внешний и внутренний компонент как взаимосвязанные пары.

Метаинтерфейсы

Интерфейсы, вызывающие конфликты внутреннего и внешнего компонента, — это обычно интерфейсы с перекрывающейся функциональностью. Если Вы можете гарантировать, что функциональность интерфейсов не перекрывается, вероятность конфликтов уменьшается. Добиться этого непросто, так как внутренний компонент может быть модернизирован для поддержки новых интерфейсов, которых внешний компонент не ожидает.

Вероятность конфликта с существующими интерфейсами компонента минимальна в случае *метаинтерфейсов (metainterfaces)*, или *интерфейсов класса*. Метаинтерфейсы манипулируют самим компонентом, а не абстракцией, которую он представляет. Предположим, что у нас есть программа преобразования растровых изображений. Пользователь может модифицировать растровое изображение с помощью различных алгоритмов. Последние реализованы как внутренний компонент, который пользователь может добавлять в систему во время работы. Каждый внутренний компонент может считывать и сохранять растровые образы, а также преобразовывать их в соответствии с некоторым особым алгоритмом. У внешнего компонента есть интерфейс *ISetColors*, устанавливающий цвета, с которыми работает внутренний компонент. Внешний компонент также имеет интерфейс *IToolInfo*, который отображает значки разных алгоритмов преобразования на панели инструментов и создает внутренний компонент,
когда пользователь выбирает соответствующий значок (рис. 9.7).

ISetColors — это пример обычного интерфейса, который расширяет абстракцию алгоритма преобразования. Компонент преобразования, вероятно, уже поддерживает интерфейс, например, *IColors*, для манипуляции набором цветов. Второй интерфейс, *IToolInfo* — пример метаинтерфейса. Все его операции служат для того, чтобы предоставить приложению способ работать с алгоритмами преобразования как с инструментами. Он не имеет никакого отношения к собственно преобразованию растровых образов. Этот интерфейс не расширяет абстракцию компонента преобразования растровых образов, но предоставляет клиенту информацию о самом этом компоненте.

Метаинтерфейсы наиболее полезны, когда они реализованы для набора разных компонентов в системе компонентов. Такие метаинтерфейсы предоставляют системе унифицированный способ работы с компонентами, повышая, таким образом, степень повторной применимости кода посредством полиморфизма. Добавление метаинтерфейсов к существующим компонентам чаще всего выполняют разработчики клиентов. Используя метаинтерфейсы, такой клиент может получать компоненты из самых разных источников и работать со всеми ними единообразно.

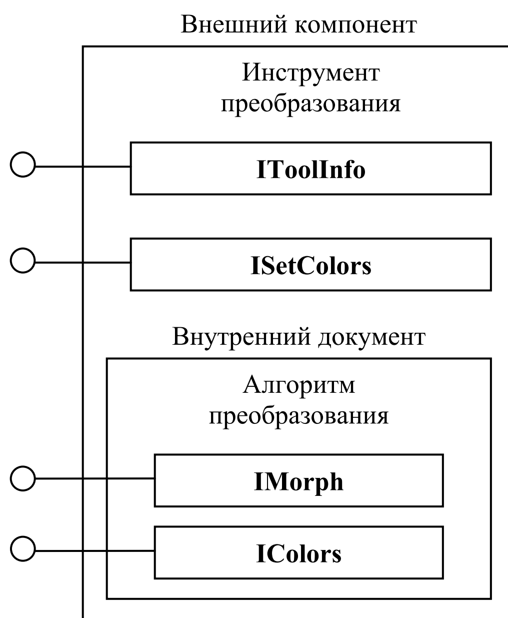


Рис. 9.7 *Интерфейс IToolInfo — это метаинтерфейс, вероятность конфликта которого с интерфейсами внутреннего компонента мала. ISetColors не является метаинтерфейсом и действует в области компетенции внутреннего компонента. ISetColors конфликтует с интерфейсом IColors внутреннего компонента.*

Взаимосвязанные пары

Другой способ избежать конфликтов интерфейсов — дать клиенту дополнительные сведения о внешнем компоненте. Если клиент и внешний компонент разрабатывались совместно, первый знает, какие интерфейсы реализованы вторым, и может использовать их вместо тех, что потенциально могут реализовываться внутренним компонентом. Если совместно разрабатывались внешний и внутренний компоненты, их можно проектировать так, чтобы конфликтных интерфейсов не было. Однако подобные взаимосвязанные пары требуют, чтобы Вы контролировали не только внешний компонент, но также и клиент либо внутренний компонент.

Агрегирование и включение в реальном мире

Мы познакомились с тем, как обеспечить повторное применение компонентов при помощи включения и агрегирования. Однако при повторном применении компонентов не все так гладко. Поскольку внешний компонент — всего лишь клиент внутреннего, он может специализировать метод последнего, вызывая другие методы только перед ним или после него. Внешний компонент не может вставить новый вариант поведения в середину такого метода. Кроме того, поскольку метод может изменять внутреннее состояние компонента, к которому у клиента нет доступа, Вы не можете заменить всю реализацию отдельного метода интерфейса. Например, если Вы используете некоторый интерфейс для открытия файла, для закрытия этого файла Вы обязаны использовать тот же самый интерфейс — поскольку Вам ничего не известно о реализации интерфейса или о какой-либо из его функций. Повторю, у внешнего компонента нет никакого дополнительного доступа или возможностей по сравнению с обычным клиентом. Внешний компонент обязан использовать внутренний совместимым и корректным образом, как и любой другой клиент.

Для устранения этих недостатков компонент можно спроектировать так, чтобы способствовать повторной применимости. То же самое верно и для классов С++, которые трудно, если не невозможно, расширять или специализировать, если они не спроектированы надлежащим образом. Классы С++, разработанные с учетом последующей настройки, имеют «защищенные» методы, которые используются производным классом для получения информации о внутреннем состоянии базового класса. Компоненты СОМ могут использовать интерфейсы для того, чтобы облегчить свое включение или агрегирование.

Предоставление информации о внутреннем состоянии

В СОМ все делается через интерфейсы. Следовательно, внутренний компонент может предоставить информацию о своем внутреннем состоянии, добавив новый интерфейс. Этот интерфейс мог бы предоставлять внешнему компоненту информацию или сервисы, помогающие настроить внутренний компонент (рис. 9.8).

В интерфейсах внутреннего состояния нет ничего загадочного. Это нормальные интерфейсы, которые предоставляют контролируемый доступ к внутреннему состоянию компонента, чтобы упростить настройку, или чтобы настройка вообще стала возможна. Обычные клиенты также при желании могут использовать интерфейсы внутреннего состояния, но в большинстве случаев эти интерфейсы будут им бесполезны.

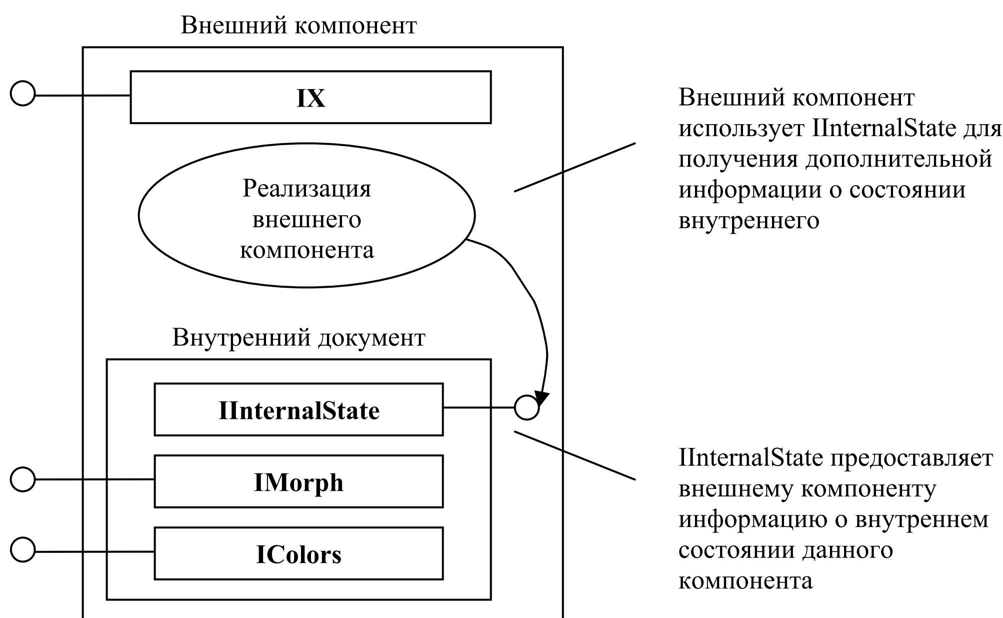


Рис. 9.8 *Внутренний компонент может упростить свою настройку, предоставив интерфейсы, которые дают внешнему компоненту доступ к его внутреннему состоянию*

Интерфейсы, предоставляющие внешнему компоненту такого рода информацию, играют в COM важную роль. Компоненты COM обычно состоят из множества небольших интерфейсов. Внутри себя реализации этих интерфейсов взаимозависимы, так как используют общие переменные-члены и другие аспекты внутреннего состояния. Это означает, что Вы не можете взять один из интерфейсов компонента и использовать его сам по себе, потому что этот интерфейс может зависеть (в управлении некоторым аспектом внутреннего состояния компонента) от информации или состояния другого интерфейса (рис. 9.9). Повторю еще раз, внешний компонент — просто клиент внутреннего. У него нет никаких особых возможностей.

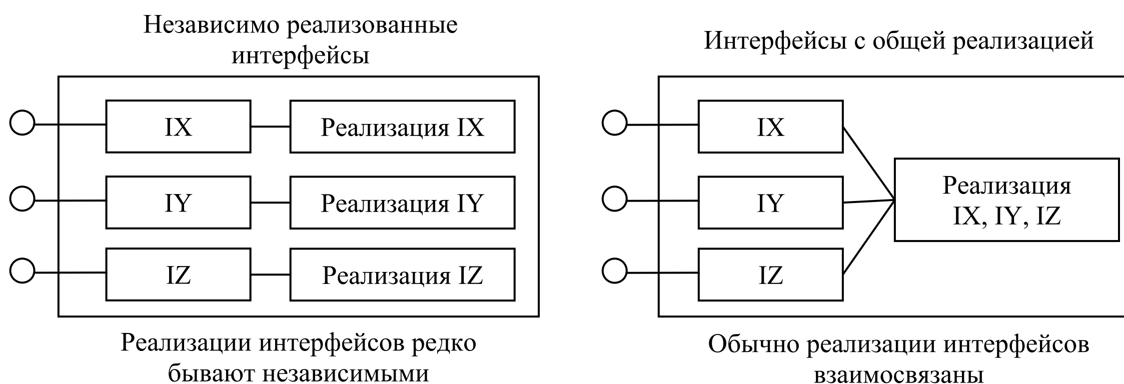


Рис. 9.9 *Клиент рассматривает интерфейсы как независимые сущности. Однако обычно интерфейсы используют общие детали реализации. Это еще более затрудняет расширение или специализацию компонентов. Добавление интерфейсов, предоставляющих взгляд на компонент изнутри, может упростить его настройку.*

Моделирование виртуальных функций

Дополнительные интерфейсы могут не только предоставить эквивалент COM для защищенных функций-членов C++, но и позволить интерфейсам замещать виртуальные

функции. Во многих случаях виртуальные функции используются как функции обратного вызова (callback). Базовый класс может вызывать виртуальную функцию до, во время или после некоторой операции, чтобы дать производному классу возможность модифицировать ее выполнение. Компоненты COM могут делать то же самое, определив интерфейс настройки (customization interface). Компонент не реализует такой интерфейс, а, наоборот, вызывает его. Клиенты, желающие настроить компонент для своих нужд, реализуют интерфейс настройки и передают указатель на него компоненту. Эту технику клиенты могут применять, и не используя включение или агрегирование (рис. 9.10).

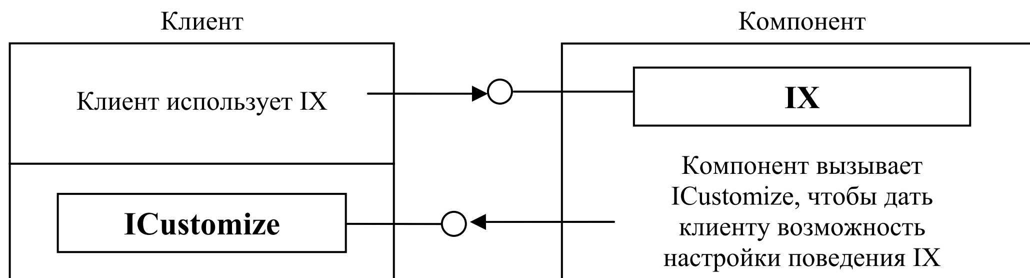


Рис. 9.10 *Компонент определяет исходящий интерфейс, который он вызывает для своей настройки*

Единственное реальное различие между наследованием и использованием функции обратного вызова или исходящего (outgoing) интерфейса, такого как *ICustomize*, состоит в том, что в последнем случае необходимо вручную присоединить клиент к компоненту. Эту технику мы рассмотрим в гл. 14. Используя ее, необходимо быть осторожным с циклическим подсчетом ссылок. В общем случае *ICustomize* реализуют как часть своего собственного компонента с отдельным счетчиком ссылок.

Хорошие компоненты предоставляют своим потенциальным пользователям информацию о внутреннем состоянии и интерфейсы настройки. По-настоящему хорошие компоненты предоставляют еще и реализацию интерфейсов настройки по умолчанию. Это экономит время на реализацию интерфейса, так как клиент может агрегировать реализацию по умолчанию.

Теперь у Вас все есть. В сущности, все, что дает наследование реализации, можно заново воссоздать с помощью интерфейсов, включения и агрегирования. Ясно, что наследование реализации C++ более привычно по сравнению с решением COM, которое представлено на рис. 9.11. Однако наследование реализации C++ требует доступа к исходному коду базового класса, а это означает, что при изменении базового класса программу придется перекомпилировать. В компонентной архитектуре, где компоненты постоянно и независимо друг от друга изменяются, а исходные коды недоступны, перекомпиляция невозможна. Делая повторное применение компонентов аналогичным простому использованию, COM ослабляет взаимозависимость между данным компонентом и компонентом, который он специализирует.

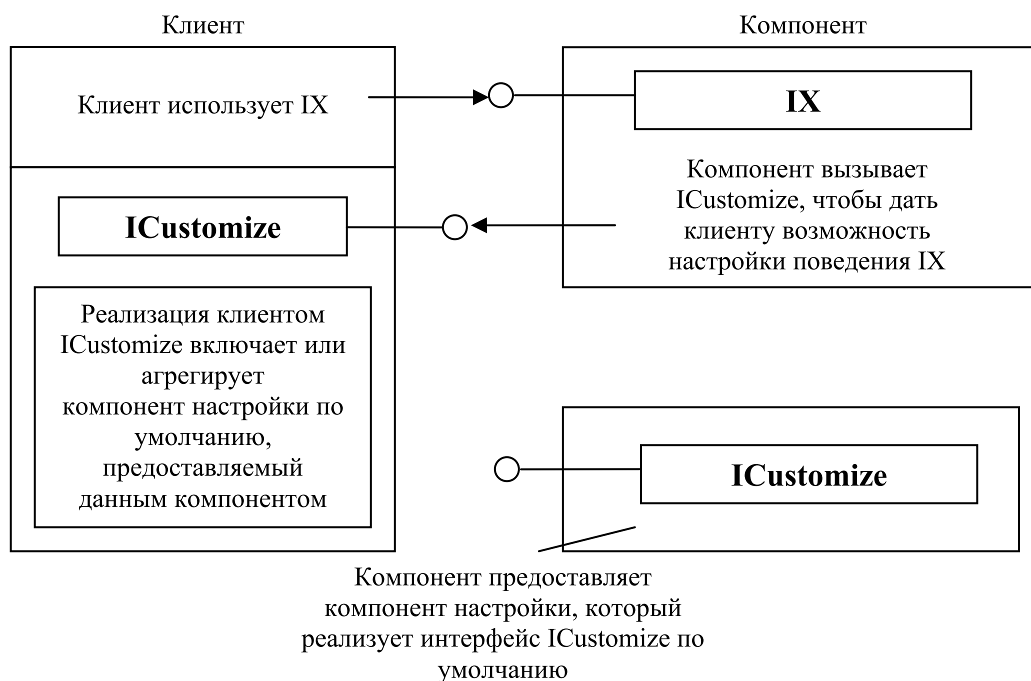


Рис. 9.11 *Компонент предоставляет реализацию исходящего интерфейса по умолчанию*

Резюме

Мы познакомились с тем, как можно повторно использовать, расширять и специализировать компоненты посредством включения и агрегирования. Повторное использование компонента столь же просто, как и включение его в другой компонент. В этом случае внешний компонент — клиент внутреннего. Если Вы хотите специализировать компонент, можете добавить свой код перед вызовом методов его интерфейсов и после него. Если Вы собираетесь только добавлять интерфейсы, можете вместо включения использовать агрегирование. Агрегирование, как я люблю говорить, — это особый вариант включения. Когда внешний компонент агрегирует интерфейс, принадлежащий внутреннему, он передает указатель на этот интерфейс непосредственно клиенту.

Внешний компонент не реализует этот интерфейс снова и не перенаправляет ему вызовы. Компонент нельзя агрегировать, если его реализация этого не поддерживает. У внутреннего компонента должно быть два разных интерфейса *IUnknown*. Один из них фактически реализует методы *IUnknown*. Другой делегирует их вызовы неделегирующему *IUnknown*, если компонент не агрегирован, и *IUnknown*, принадлежащий внутреннему компоненту, если агрегирован.

Включение и агрегирование предоставляют надежный механизм для повторного использования и настройки компонентов. Они устраняют необходимость применять в архитектурах СОМ наследование реализации. Наследование реализации нежелательно в системах, где клиенты должны быть изолированы от реализации используемых ими компонентов. Если клиент не изолирован от деталей реализации компонента, при изменении компонента его придется переписать, перекомпилировать или перекомпоновать.

В этой главе мы научились повторному применению компонентов. В следующей главе будет рассмотрен другой вариант: вместо повторного применения компонентов мы будем повторно применять код на С++. Мы реализуем базовые классы для *IUnknown* и *IClassFactory*, от которых смогут наследоваться наши компоненты. От всех этих разговоров о повторном применении у меня разыгрался аппетит. Наверное, пора узнать, насколько на самом деле съедобны миллионеры. Говорят, они хрустят на зубах.

10. Будем проще

Что бы ни делал человек, он пытается максимально упростить задачу — особенно, кажется, это касается любых учебных упражнений, которые в результате могут полностью лишиться смысла. Телевидение наводняет реклама — и, похоже, успешная — тренажера для брюшного пресса и тому подобных приспособлений. Такой тренажер механизирует одно из простейших упражнений, облегчая его выполнение до предела. (Как говорят, главное достоинство тренажера в том, что он позволяет сконцентрировать усилия на мышцах живота, а не шеи или чего-нибудь еще. Как знать…)

Поскольку все хотят упростить себе жизнь, я поддамся общей тенденции и упрощу использование компонентов COM. Писать их сложнее, чем тренировать мышцы живота, и, надо сказать, компоненты не становятся ни лучше, ни красивее, если на их создание потратить больше усилий.

Сначала при помощи классов C++ мы сделаем использование компонентов COM более похожим на использование классов; затем разработаем несколько классов, облегчающих разработку компонентов.

Упрощения на клиентской стороне

Большинство из Вас не надо убеждать, что использовать компоненты COM вовсе не так просто, как обычные классы C++. Во-первых, необходимо подсчитывать ссылки. Если Вы забыли вызвать *AddRef* для указателя на интерфейс, можете сразу прощаться с выходными. Если ссылки подсчитываются неправильно, программа может попытаться работать с интерфейсом уже удаленного компонента, что кончится сбоем. Найти пропущенный вызов *AddRef* или *Release* нелегко. Хуже того, при каждом новом запуске программы компонент может освобождаться в разных точках. Хотя мне доставляетнастоящее удовольствие отлавливать трудновоспроизводимые ошибки (в обществе нескольких друзей и пиццы), не многие, кажется, разделяют эту радость.

Поддержка COM в компиляторе

Компилятор Microsoft Visual C++ версии 5.0 вводит расширения языка C++, упрощающие разработку и использование компонентов COM. Для получения более подробной информации обратитесь к документации Visual C++ версии 5.0.

Даже если Вы вписали вызовы *Release* там, где нужно, Ваша программа может их не выполнить. Обработчики исключений C++ ничего не знают о компонентах COM. Поэтому *Release* не вызывается автоматически после возникновения исключений.

Простая и корректная обработка *AddRef* и *Release* — лишь полдела. Нужно еще упростить вызов *QueryInterface*, Вы, я уверен, давно заметили, что один такой вызов занимает несколько строк. Несколько вызовов внутри одной функции могут легко затенить содержательный код. Я обхожу эту проблему, сохраняя указатели и интерфейсы, а не запрашивая их при всякой нужде. Это повышает производительность и надежность кода — за счет памяти. Но с *QueryInterface* связана и более тяжелая проблема — вызов требует явного приведения типов. Если Вы перепутаете параметры, передаваемые *QueryInterface*, компилятор Вам не поможет. Например, следующий код прекрасно компилируется, хотя он и помещает указатель на интерфейс *IY* в переменную-указатель на *IZ*:

```
IZ* pIZ;
PIX->QueryInterface(IID_IY, (void**)&pIZ);
```

Зловредный указатель *void* снова подставляет нам ножку, скрывая типы от нашего любимого компилятора. Эти проблемы можно устранить при помощи инкапсуляции. Можно либо инкапсулировать указатель на интерфейс при помощи класса smart-указателя, либо инкапсулировать сам интерфейс внутри класса-оболочки (wrapper). Давайте рассмотрим эти методы, начиная со smart-указателей.

Smart-указатели на интерфейсы

Первый способ упрощения кода клиента — использование для доступа к компонентам smart-указателей вместо обычных указателей на интерфейс. Smart-указатель используется так же, как обычный указатель C++, но он скрывает подсчет ссылок. Когда поток управления выходит из области действия переменной-smart-указателя, интерфейс автоматически освобождается. Это делает использование интерфейса COM аналогичным использованию объекта C++.

Что такое smart-указатель?

Smart-указатель (smart pointer) — это класс, переопределяющий operator-> (оператор выбора метода). Класс smart-указателя содержит указатель на другой объект. Когда для smart-указателя вызывается operator->, этот вызов делегируется или передается smart-указателем объекту, на который ссылается содержащийся в нем указатель. Smart-указатель на интерфейс — это smart-указатель, содержащий указатель на интерфейс.
Рассмотрим простой пример. *CFooPointer* имеет минимум возможностей, необходимых классу smart-указателя. Он содержит указатель и переопределяет *operator->*.

```
class CFoo
{
public:
  virtual void Bar();
};

class CFooPointer
{
public:
  CFooPointer (Cfoo*p) { m_p = p; }
  CFoo* operator->() { return m_p; }
private:
  CFoo* m_p;
};
...

void Funky(CFoo* pFoo)
{
  // Создать и инициализировать smart-указатель
  CFooPointer spFoo(pFoo);
  // Следующий оператор эквивалентен pFoo->Bar();
  spFoo->Bar();
}
```

В приведенном примере функция *Funky* создает *CFooPointer* с именем *spFoo* и инициализирует его с помощью *pFoo*. Затем она выполняет разыменование *spFoo* для вызова функции *Bar*. Указатель *spFoo* делегирует этот вызов *m_p,* которая содержит *pFoo*. С помощью *spFoo* можно вызвать любой метод *CFoo*. Самое замечательное здесь то, что Вам не нужно явно перечислять в *CFooPointer* все методы *CFoo.* Для *CFoo* функция

operator-> означает «разыменуй меня». В то же время для *CFooPointer* она означает «разыменуй не меня, а *m_p*» (см. рис. 10.1).

Для умного (smart) указателя *CFooPointer* глуповат. Он ничего не делает. Хороший компилятор, вероятно, вообще его устранит во время оптимизации. Кроме того, *CFooPointer* не слишком похож на обычный указатель. Попробуйте присвоить *pFoo* переменной *spFoo*. Присваивание не сработает, так как *operator=* (оператор присваивания) не переопределен соответствующим образом. Для того, чтобы *CFooPointer* выглядел так же, как и указатель *CFoo,* для *CFooPointer* необходимо переопределить несколько операторов. Сюда входят *operator** (оператор разыменования) и *operator&* (оператор получения адреса), которые должны работать с указателем *m_p*, а не с самим объектом *CFooPointer.*

Реализация класса указателя на интерфейс

Хотя классов smart-указателей для работы с интерфейсами COM и не так много, как классов строк, но число тех и других не сильно различается. ActiveX Template Library (ATL) содержит классы указателей на интерфейсы COM *CComPtr* и *CComQIPtr*. В библиотеке MFC имеется класс *CIP* для внутреннего пользования. (Он находится в файле AFXCOM_.H.) *CIP* — это самый полный вариант класса smart-указателя на интерфейс. Он делает практически все. Здесь я представлю свой собственный, главным образом потому, что мой код легче читается

Мой класс похож на классы из ATL и MFC, но не столь полон.

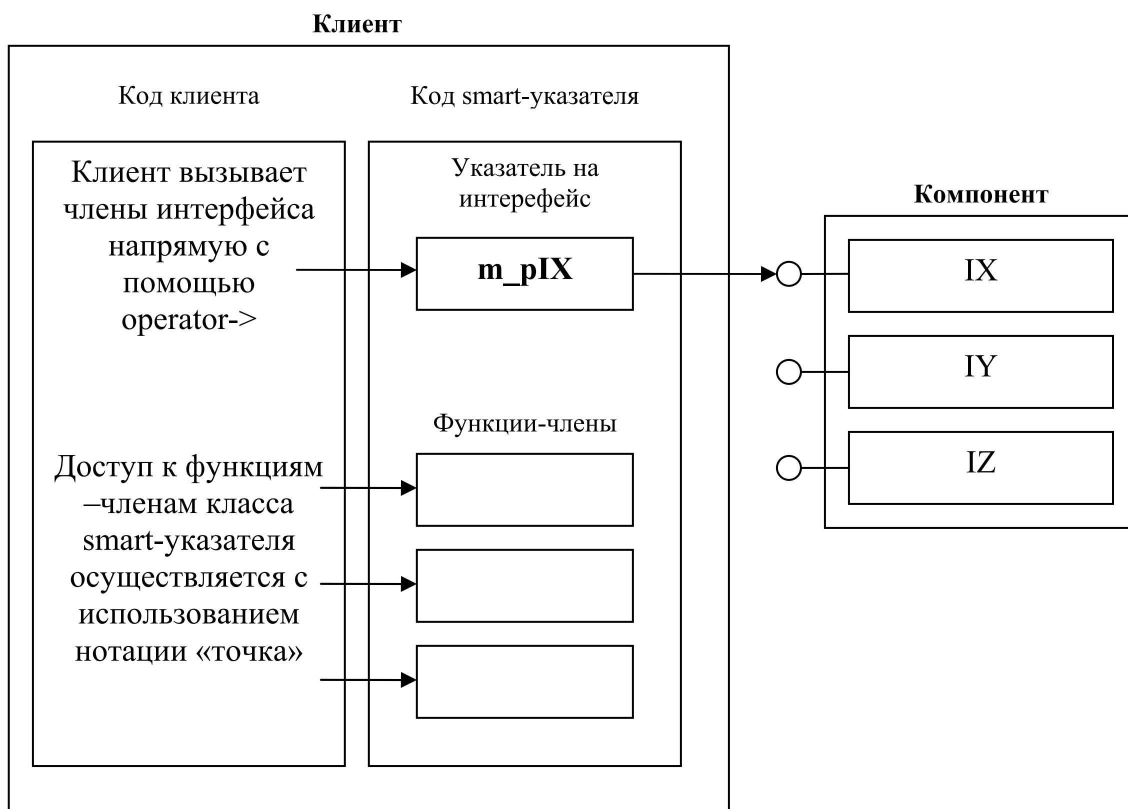


Рис. 10.1 *Smart-указатели на интерфейсы делегируют вызовы указателю на интерфейс, хранящемуся внутри класса.*

Мой класс указателя на интерфейс называется *IPtr* и реализован в файле PTR.H, который представлен в листинге 10.1. Пусть длина исходного текста Вас не пугает. Кода там очень мало. Я просто вставил побольше пустых строк, чтобы легче было читать.

Шаблон* IPtr *из PTR.H

```
//
// IPtr - Smart-указатель на интерфейс
// Использование: IPtr<IX, &IID_IX> spIX;
// Не используйте с IUnknown; IPtr<IUnknown, &IID_IUnknown>
// не будет компилироваться. Вместо этого используйте IUnknownPtr.
//
template <class T, const IID* piid> class IPtr
{
public:
  // Конструкторы
  IPtr()
  {
    m_pI = NULL;
  }
  IPtr(T* lp)
  {
    m_pI = lp;
    if (m_pI != NULL)
    {
      m_pI->AddRef();
    }
  }
  IPtr(IUnknown* pI)
  {
    m_pI = NULL;
    if (pI != NULL)
    {
      pI->QueryInterface(*piid, (void **)&m_pI);
    }
  }

  // Деструктор
  ~IPtr()
  {
    Release();
  }

  // Сброс в NULL
  void Release()
  {
    if (m_pI != NULL)
    {
      T* pOld = m_pI;
      m_pI = NULL;
      pOld->Release();
    }
  }

  // Преобразование
  operator T*() { return m_pI; }

// Операции с указателем
  T& operator*() { assert(m_pI != NULL); return *m_pI; }
  T** operator&() { assert(m_pI == NULL); return &m_pI; }
  T* operator->() { assert(m_pI != NULL); return m_pI; }
```

```
  // Присваивание того же интерфейса
  T* operator=(T* pI)
  {
    if (m_pI != pI)
    {
      IUnknown* pOld = m_pI; // Сохранить старое значение
      m_pI = pI; // Присвоить новое значение
      if (m_pI != NULL)
      {
        m_pI->AddRef();
      }
      if (pOld != NULL)
      {
        pOld->Release(); // Освободить старый интерфейс
      }
    }
    return m_pI;
  }

  // Присваивание другого интерфейса
  T* operator=(IUnknown* pI)
  {
    IUnknown* pOld = m_pI; // Сохранить текущее значение
    m_pI == NULL ;
    // Запросить соответствующий интерфейс
    if (pI != NULL)
    {
      HRESULT hr = pI->QueryInterface(*piid, (void**)&m_pI);
      assert(SUCCEEDED(hr) && (m_pI != NULL));
    }
    if (pOld != NULL)
    {
      pOld->Release(); // Освободить старый указатель
    }
    return m_pI;
  }

  // Логические функции
  BOOL operator!() { return (m_pI == NULL) ? TRUE : FALSE; }

  // Требует компилятора, поддерживающего BOOL
  operator BOOL() const
  {
    return (m_pI != NULL) ? TRUE : FALSE;
  }

  // GUID
  const IID& iid() { return *piid; }
private:
  // Хранимый указатель
  T* m_pI;
};
```

Листинг 10.1. *Класс smart-указателей на интерфейсы IPtr.*

Использование класса указателя на интерфейс

Использовать экземпляр *IPtr* легко, особенно для шаблона класса. Во-первых, Вы создаете указатель, передавая тип интерфейса и указатель на его IID. (Неплохо было бы использовать как параметр шаблона ссылку, но большинство компиляторов этого не допускают.) Теперь можно вызвать *CoCreateInstance* для создания компонента и получения указателя на него. В приведенном далее примере Вы можете видеть, насколько

эффективно *IPtr* эмулирует настоящий указатель. Мы можем использовать с *IPtr* *operator&* как будто это настоящий указатель:

```
void main()
{
  IPtr<IX, &IID_IX> spIX;
  HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                        NULL,
                        CLSCTX_ALL,
                        spIX.iid(),
                        (void**)&spIX);
  if (SUCCEEDED(hr))
  {
    spIX->Fx();
  }
}
```

Предыдущий вызов *CoCreateInstance* небезопасен в смысле приведения типов, но его можно подправить, определив в шаблоне *IPtr* другую функцию:

```
HRESULT CreateInstance(const CLSID& clsid, IUnkown* pI, DWORD clsctx)
{
  Release();
  return CoCreateInstance(clsid, pI, clsctx, *piid, (void**)&m_pI);
}
```

Ее можно было бы использовать так:

```
IPtr<IX, &IID_IX> spIX;
HRESULT hr = spPX.CreateInstance(CLSID_Component1, NULL,
CLSCTX_INPROC_SERVER);
```

Между прочим, для своих переменных-smart-указателей я использую префикс *sp*, что позволяет отделить smart-указатели от тех, что не наделены интеллектом.

Подсчет ссылок

Самое замечательное в примере со smart-указателями то, что нам не нужно помнить о вызовах *Release*. Когда поток управления выходит из области действия smart-указателя, деструктор последнего автоматически вызывает *Release*. Кроме того, интерфейс будет автоматически освобожден и в случае возникновения исключения, так как smart-указатель — это объект C++.

Если Вы хотите освободить указатель на интерфейс, хранящийся в smart-указателе, не следует вызывать *Release*. Smart-указатель ничего не знает о функциях, которые вызываются с его помощью, а слепо делегирует им вызовы. Поэтому, если освободить интерфейс, в smart-указателе по-прежнему будет храниться не NULL. Попытка использовать такой smart-указатель приведет к нарушению защиты памяти.

Для разных smart-указателей есть свои способы сообщить им, что Вы хотите освободить интерфейс. Большинство, включая *IPtr*, реализуют функцию *Release*, которая вызывается с использованием нотации «точка», а не через *operator->*:

```
spIX.Release();
```

Другой способ освобождения интерфейса в *IPtr* — присвоить ему значение NULL:

```
spIX = NULL;
```

Чтобы понять, почему это работает, рассмотрим, как *IPtr* переопределяет оператор присваивания.

Присваивание

Класс *IPtr* переопределяет *operator=*, чтобы указатель на интерфейс можно было присваивать smart-указателю:

```
T* operator=(T* pI)
{
  if (m_pI != pI)
  {
    IUnknown* pOld = m_pI;
    m_pI = pI;
    if (m_pI != NULL)
    {
      m_pI->AddRef();
    }
    if (pOld != NULL)
    {
      pOld->Release();
    }
  }
  return m_pI;
}
```

Обратите внимание на два интересных момента реализации *operator=*. Во-первых, она вызывает *AddRef* и *Release*, так что нам не нужно делать это самостоятельно. Во-вторых, smart-указатель на интерфейс освобождает свой текущий указатель *после* присваивания нового. Это предотвращает удаление компонента из памяти до присваивания.

Представленный ниже фрагмент кода присваивает значение указателя *pIX1* члену *m_p* переменной *spIX*. По ходудела операция присваивания вызывает для указателя *AddRef*. После использования *spIX* ей присваивается *pIX2*. Переопределенная функция *operator=* копирует во временную переменную текущий указатель, который указывает на интерфейс *pIX1*, сохраняет *pIX2* и вызывает *AddRef* для *pIX2*.

```
void Fuzzy(IX* pIX1, IX* pIX2)
{
  IPtr<IX, &IID_IX> spIX;
  spIX=pIX1;
  spIX->Fx();
  spIX = pIX2;
  spIX->Fx();
}
```

Я определил операцию преобразования так, чтобы Вы могли присваивать один объект *IPtr* другому объекту *Iptr* того же типа. Пример подобного присваивания приведен ниже:

```
typedef IPtr<IX, &IID_IX> SPIX;
SPIX g_spIX;

void Wuzzy(SPIZ spIX)
{
  g_spIX = spIX;
}
```

Эта операция присваивания работает, только если оба указателя имеют один тип. Обратите внимание на использование *typedef* для повышения читабельности кода.

Присваивание *IUnknown*

Как Вы помните, одной из наших целей было упростить вызов *QueryInterface*. Это можно сделать при помощи еще одной совмещенной операции присваивания.

```
T* operator=(IUnknown* pIUnknown);
```

Если Вы присваиваете указатель на интерфейс, тип которого отличен от типа smart-указателя, операция присваивания автоматически вызовет *QueryInterface*. Например, в следующем далее фрагменте кода указатель на *IY* присваивается smart-указателю на *IX*. Не забывайте проверить значение указателя, чтобы убедиться в том, что присваивание было успешным. Некоторые классы smart-указателей генерируют исключение, если вызов *QueryInterface* заканчивается ошибкой.

```
void WasABear(IY* pIY)
{
  IPtr<IX, &IID_IX> spIX;
  spIX = pIY;
  if (spIX)
  {
    spIX->Fx();
  }
}
```

Лично мне не нравится операция присваивания, вызывающая *QueryInterface*. Одно из правил переопределения операций рекомендует, чтобы переопределенная версия вела себя аналогично обычной, встроенной. Очевидно, что в случае операции присваивания с вызовом *QueryInterface* это не так. Оператор присваивания C++ всегда выполняется успешно. Но вызов *QueryInterface* может закончиться неудачей, поэтому и вызывающая его операция присваивания тоже может не сработать. К сожалению, вся индустрия информатики в этом вопросе против меня. Microsoft Visual Basic содержит операцию присваивания, которая вызывает *QueryInterface*. Smart-указатели на интерфейсы в ATL и MFC также переопределяют операцию присваивания, чтобы вызывать *QueryInterface*.

interface_cast

Я действительно не люблю скрывать значительные объемы своего кода за невинно выглядящими присваиваниями. То, что Visual Basic вызывает во время присваивания *QueryInterface*, не означает, что и в C++ правильно и целесообразно так делать. Может быть, в смысле синтаксиса C++ — это дурной сон, но тогда Visual Basic — ночной кошмар.

Я предпочитаю инкапсулировать *QueryInterface* при помощи функции, которую назвал *interface_cast*. *interface_cast* — это функция-шаблон, которая используется аналогично *dynamic_cast*. Вот она:

```
template <class I, const GUID* pGUID>
I* interface_cast(IUnknown* pIUnknown)
{
  I* pI = NULL;
  HRESULT hr = pIUnknown->QueryInterface(*pGUID, (void**)&pI);
  assert(SUCCEEDED(hr));
  return pI;}
```

Используется *interface_cast* так:

```
IY* pIX = interface_cast<IY, &&IID_IY>(this);
```

Приятная особенность *interface_cast* состоит в том, что для ее использования даже не нужны smart-указатели. Плохая же сторона в том, что Вам потребуется компилятор, который поддерживает явную генерацию экземпляров (explicit instantiation) функций-шаблонов. По счастью, Visual C++ версии 5.0 — именно такой компилятор.

IUnknownPtr

В дополнение к *IPtr* в PTR.H находится еще один класс указателя — *IUnknownPtr* — версия *IPtr*, предназначенная для использования с *IUnknown*. Класс *IUnknownPtr* не является шаблоном и не реализует операцию присваивания, вызывающую *QueryInterface*. Я создал *IUnknownPtr* потому, что *IPtr* нельзя использовать с *IUnknown* в качестве параметра шаблона. Попытка сгенерировать вариант *IPtr* для *IUnknown* приведет к порождению двух операций присваивания с одинаковыми прототипами. Не используйте *IPtr* для объявления smart-указателя на *IUnknown*:

```
IPtr<IUnknown, &IID_IUnknown> spIUnknown; // Ошибка
```

Вместо этого применяйте *IUnknownPtr*:

```
IUnknownPtr spIUnknown;
```

Реализация клиента с помощью smart-указателей

В примеры этой главы, находящиеся на прилагаемом к книге диске, входит код двух клиентов. Клиент 1 реализован так же, как в предыдущих главах. Клиент 2 — через smart-указатели. В листинге 10.2 показан код Клиента 2. Очевидно, когда все вызовы *QueryInterface* скрыты, код читается гораздо легче. Как обычно, с помощью *typedef* использование классов шаблонов можно сделать еще удобнее. Я поместил весь код, который работает с компонентом, в функцию *Think*.

CLIENT2.CPP

```
//
// Client2.cpp - Реализация клиента с помощью smart-указателей
//

#include <objbase.h>

#include "Iface.h"
#include "Util.h" // Трассировка с метками
#include "Ptr.h" // Классы smart-указателей

static inline void trace(const char* msg)
{ Util::Trace("Клиент 2", msg, S_OK); }

static inline void trace(const char* msg, HRESULT hr)
{ Util::Trace("Клиент 2", msg, hr); }
```

```
void Think()
{
  trace("Создать Компонент 1");
  IPtr<IX, &IID_IX> spIX;
  HRESULT hr = CoCreateInstance(CLSID_Component1,
                        NULL,
                        CLSCTX_INPROC_SERVER,
                        spIX.iid(),
                        (void**)&spIX);

  if (SUCCEEDED(hr))
  {
    trace("Компонент успешно создан");
    spIX->Fx();
    trace("Получить интерфейс IY");
    IPtr<IY, &IID_IY> spIY;
    spIY = spIX; // Используется присваивание
    if (spIY)
    {
      spIY->Fy();
      trace("Получить интерфейс IX через IY");
      IPtr<IX, &IID_IX> spIX2(spIY); // Используется конструктор
      if (!spIX2)
      {
        trace("Не могу получить интерфейс IX через IY");
      }
      else
      {
        spIX2->Fx();
      }
    }
    trace("Получить интерфейс IZ");
    IPtr<IZ, &IID_IZ> spIZ;
    spIZ = spIX;
    if (spIZ)
    {
      spIZ->Fz();
      trace("Получить интерфейс IX через IZ");
      IPtr<IX, &IID_IX> spIX2(spIZ);
      if (!spIX2)
      {
        trace("Не могу получить интерфейс IX через IZ");
      }
      else
      {
        spIX2->Fx();
      }
    }
  }
  else
  {
    trace("Не могу создать компонент", hr);
  }
}
int main()
{
  // Инициализировать библиотеку COM
  CoInitialize(NULL);
  Think();              // Упражнения со smart-указателями
  CoUninitialize();     // Освободить библиотеку COM
  return 0;
}
```
Листинг 10.2 *Клиент 2 использует класс smart-указателей на интерфейсы IPtr*

Обратите внимание, как в примере обрабатываются возможные ошибки инкапсулированных *QueryInterface*. Для проверки успешного завершения мы используем функцию преобразования в тип BOOL, которая возвращает TRUE, если *IPtr::m_p* есть не NULL. Для проверки на ошибку применяется *operator!*.

Проблемы smart-указателей

Большинство этих проблем незначительно. Вам необходимо избегать вызова *Release* для smart-указателей на интерфейсы. Доступ к методам класса smart-указателя осуществляется с помощью «точки», а не «стрелки», которая используется для доступа к методам интерфейса. Все эти проблемы компенсируются необычайной гибкостью классов smart-указателей. Вы можете написать один такой класс, который будет работать для всех интерфейсов (кроме *IUnknown*). Однако это свойство smart-указателей является их главным недостатком. Smart-указатели настолько универсальны, что инкапсулируют не используемый Вами интерфейс, но использование указателей на интерфейс. Для большинства простых интерфейсов это идеально. Но в некоторых случаях лучше инкапсулировать сам интерфейс.

Классы-оболочки C++

Smart-указатели прекрасно подходят, когда Вы хотите инкапсулировать группу интерфейсов. Для инкапсуляции конкретного интерфейса используйте класс-оболочку C++. *Класс-оболочка (wrapper class)* — это просто клиент одного или нескольких интерфейсов COM, предоставляющий абстракцию использования этих интерфейсов. Ваша программа вызывает методы класса-оболочки, которые вызывают методы интерфейса COM. Классы-оболочки упрощают вызовы интерфейсов COM, подобно тому, как классы-оболочки MFC упрощают работу с Win32 (рис. 10.2).

Важнейшая отличительная черта классов-оболочек — то, что они могут использовать такие средства C++, как совмещение имен функций, совмещение операторов и параметры по умолчанию; благодаря этому программиста на C++ может работать с классами-оболочками привычным способом. В Visual C++ имеется инструмент, автоматически генерирующий классы-оболочки для управляющих элементов ActiveX и многих других компонентов COM.

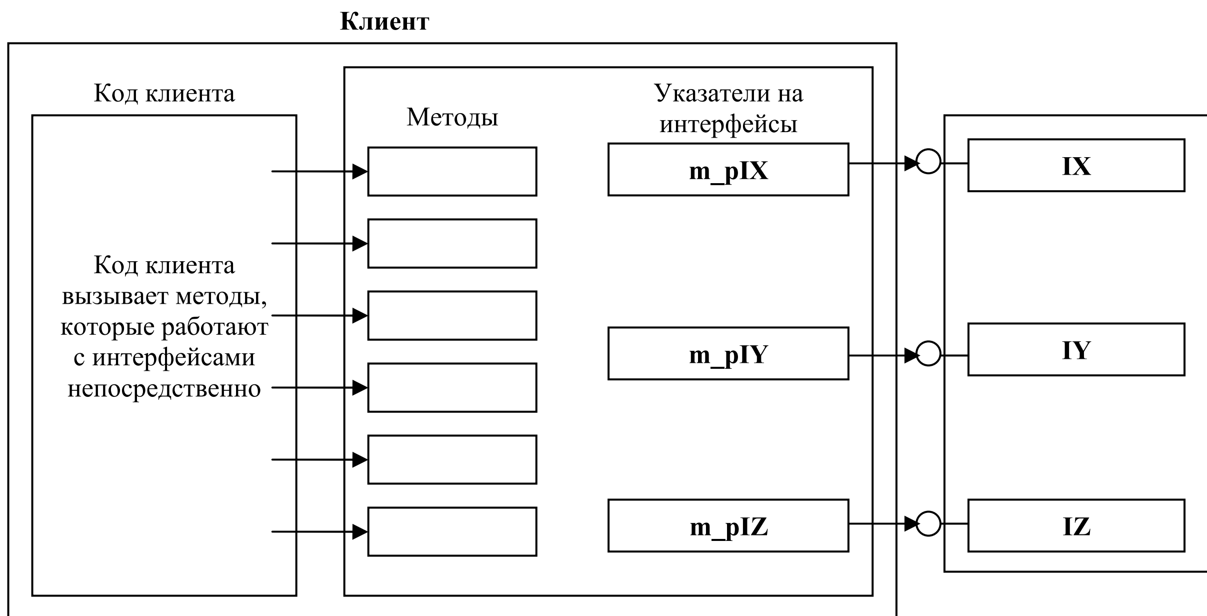


Рис. 10.2 *Классы-оболочки могут инкапсулировать интерфейсы компонента и значительно облегчить работу с ним*

Классы-оболочки — аналог включения…

В отличие от smart-указателей, классы-оболочки должны повторно реализовывать все функции интерфейсов, оболочками которых они являются — даже в том случае, если они не добавляют к самим интерфейсам никакой новой функциональности. Другое существенное различие между классами-оболочками и smart-указателями — то, что первые могут добавить новый код перед вызовом функции интерфейса и после него. Если Вы сравните это с приемами повторного применения компонентов из предыдущей главы, то станет очевидно, что классы-оболочки являются аналогами включения, тогда как smart-указатели — агрегирования.

Оболочки нескольких интерфейсов

Классы-оболочки используют также для объединения в один логический блок нескольких интерфейсов. Как Вы помните, у компонентом СОМ обычно много маленьких интерфейсов. Наличие множества интерфейсов дает простор для полиморфного использования компонентов, что подразумевает большую вероятность повторного применения архитектуры. Хотя маленькие интерфейсы прекрасно подходят для повторного использования, в первый раз их трудно использовать. В случае маленьких интерфейсов Вам может понадобиться один интерфейс для сгибания объекта, второй для его распрямления и совершенно иной, чтобы его смять. Это может повлечь множество запросов интерфейсов и соответствующие накладные расходы. Ситуация походит на общение с бюрократической структурой. У каждого бюрократа есть своя строго определенная задача, и с любым вопросом Вам придется обойти множество чиновников. Объединение всех интерфейсов в один класс C++ упрощает использование объектов с данным набором интерфейсов. Поддержка OLE в MFC, по сути дела, осуществляется большим классом-оболочкой.

Используете ли Вы smart-указатели, классы-оболочки или smart-указатели внутри классов-оболочек — эти приемы программирования могут облегчить отладку и чтение кода Вашего клиента.

Упрощения на серверной стороне

Теперь пора познакомиться с тем, как упростить реализацию компонентов СОМ. В предыдущем разделе упрощалось *использование* компонентов. В этом разделе мы собираемся упростить их *реализацию*. Для этого мы предпримем атаку с двух направлений. Первым будет *CUnknown* — базовый класс, реализующий *IUnknown*. Если Ваш класс наследует *CUnknown*, Вам не нужно беспокоиться о реализации *AddRef* и *Release*, а реализация *QueryInterface* упрощается.

Второе направление атаки — это реализация *IClassFactory*, которая будет работать с любым компонентом СОМ, производным от *CUnknown*. Для того, чтобы *CFactory* могла создать компонент, достаточно просто поместить CLSID и другую информацию в некую структуру данных. *CFactory* и *CFactory* — очень простые классы, и Вы легко поймете их исходные тексты. Поэтому основное внимание я собираюсь уделить использованию *Cunknown* и *CFactory* для реализации компонента СОМ, а не реализации самих классов. Тем не менее, мы начнем с краткого обзора *CUnknown* и *CFactory*. Эти классы будут использоваться в примерах оставшихся глав книги, что и было основной причиной их создания — я хотел сократить себе объем работы.

Базовый класс *CUnknown*

В гл. 9 мы видели, что агрегируемому компоненту нужны два интерфейса *IUnknown*: делегирующий и неделегирующий. Если компонент агрегируется, делегирующий *IUnknown* делегирует вызовы *Iunknown* внешнего компонента. В противном случае он делегирует их неделегирующему *IUnknown*. Поскольку мы хотим поддерживать компоненты, которые могут быть агрегированы, наш *CUnknown* должен реализовывать *InondelegatingUnknown*, а не *IUnknown*. Заголовочный файл *CUnknown* показан в листинге 10.3.

CUNKNOWN.H

```
#ifndef __CUnknown_h__
#define __CUnknown_h__
#include <objbase.h>

///////////////////////////////////////////////////////////
//
// Неделегирующий интерфейс IUnknown
// - Неделегирующая версия IUnknown
//
interface INondelegatingUnknown
{
  virtual HRESULT __stdcall
  NondelegatingQueryInterface(const IID& iid, void** ppv) = 0;
  virtual ULONG __stdcall NondelegatingAddRef() = 0;
  virtual ULONG __stdcall NondelegatingRelease() = 0;
};

///////////////////////////////////////////////////////////
//
// Объявление CUnknown
// - Базовый класс для реализации IUnknown
//
class CUnknown : public INondelegatingUnknown
{
public:
  // Реализация неделегирующего IUnknown
  virtual HRESULT __stdcall NondelegatingQueryInterface(const IID&, void**);
  virtual ULONG __stdcall NondelegatingAddRef();
  virtual ULONG __stdcall NondelegatingRelease();
```

```
  // Конструктор
  CUnknown(IUnknown* pUnknownOuter);

  // Деструктор
  virtual ~CUnknown();

  // Инициализация (особенно важна для агрегатов)
  virtual HRESULT Init() { return S_OK; }

  // Уведомление производным классам об освобождении объекта
  virtual void FinalRelease();

  // Текущий счетчик активных компонентов
  static long ActiveComponents() { return s_cActiveComponents; }

  // Вспомогательная функция
  HRESULT FinishQI(IUnknown* pI, void** ppv);
protected:


  // Поддержка делегирования
  IUnknown* GetOuterUnknown() const
  {return m_pUnknownOuter ;}
private:

  // Счетчик ссылок данного объекта
  long m_cRef;
  // Указатель на внешний IUnknown
  IUnknown* m_pUnknownOuter;
  // Счетчик общего числа активных компонентов
  static long s_cActiveComponents;
};

///////////////////////////////////////////////////////////
//
// Делегирующий IUnknown
// - Делегирует неделегирующему IUnknown или внешнему
// IUnknown, если компонент агрегируется
//

#define DECLARE_IUNKNOWN                                    \
virtual HRESULT __stdcall                                   \
QueryInterface(const IID& iid, void** ppv)                  \
{                                                           \
  return GetOuterUnknown()->QueryInterface(iid,ppv);        \
};                                                          \

virtual ULONG __stdcall AddRef()                            \
{                                                           \
  return GetOuterUnknown()->AddRef();                       \
};                                                          \

virtual ULONG __stdcall Release()                           \
{                                                           \
  return GetOuterUnknown()->Release();                      \
};
///////////////////////////////////////////////////////////
#endif
```

Листинг 10.3 *Базовый класс CUnknown реализует неделегирующий IUnknown. Макрос DECLARE_IUNKNOWN реализует делегирующий IUnknown*

Реализация интерфейса *INondelegatingUnknown* в классе *CUnknown* аналогична той, что была дана для агрегируемого компонента в гл. 8. Конечно, *CUnknown* не может заранее знать, какие интерфейсы будут реализованы производными компонентами. Как мы увидим ниже, чтобы добавить код для предоставляемых ими интерфейсов, компоненты должны переопределить функцию *NondelegatingQueryInterface*.

Реализация *CUnknown* находится в файле CUNKNOWN.CPP на прилагающемся к книге диске. Мы не будем изучать этот код целиком. Однако позвольте кратко рассмотреть еще несколько важных моментов.

Макрос *DECLARE_IUNKNOWN*

Я просто не хочу реализовывать делегирующий *IUnknown* всякий раз, когда нужно реализовать компонент. В конце концов, именно поэтому и появился *CUnknown*. Обратившись к листингу 10.3, Вы увидите, в конце макрос *DECLARE_IUNKNOWN*, реализующий делегирующий *IUnknown*. Да, я помню, что ненавижу макросы, но чувствую, что здесь это именно то, что нужно. ActiveX Template Library избегает использования обычных макросов; взамен применяются проверяемые компилятором макросы, более известные как шаблоны.

Еще одна причина использования *INondelegatingUnknown*

Поддержка агрегирования — достаточная причина сама по себе; но есть и еще одна причина, по которой *CUnknown* реализует *INondelegatingUnknown*, а не *IUnknown*: производный класс обязан реализовать любой абстрактный базовый класс, который наследует. Предположим, что мы используем в качестве абстрактного базового класса *IUnknown*. Класс *CA* наследует *IUnknown* и должен реализовать его чисто виртуальные функции. Но мы хотим использовать существующую реализацию *IUnknown* — *CUnknown* — повторно. Если *CA* наследует и *CUnknown*, и *IUnknown*, он по-прежнему обязан реализовать чисто виртуальные функции *IUnknown*.

Теперь предположим, что *CA* наследует *IUnknown* и через *CUnknown*, и через *IX*. Поскольку *IX* не реализует функции-члены *IUnknown*, они по-прежнему остаются абстрактными, и *CA* обязан их реализовать. В данном случае реализация функций в *CA* скрыла бы их реализацию в *CUnknown*. Во избежание этой проблемы наш *CUnknown* реализует *INondelegatingUnknown*. Правильный вариант изображен на рис. 10.3.

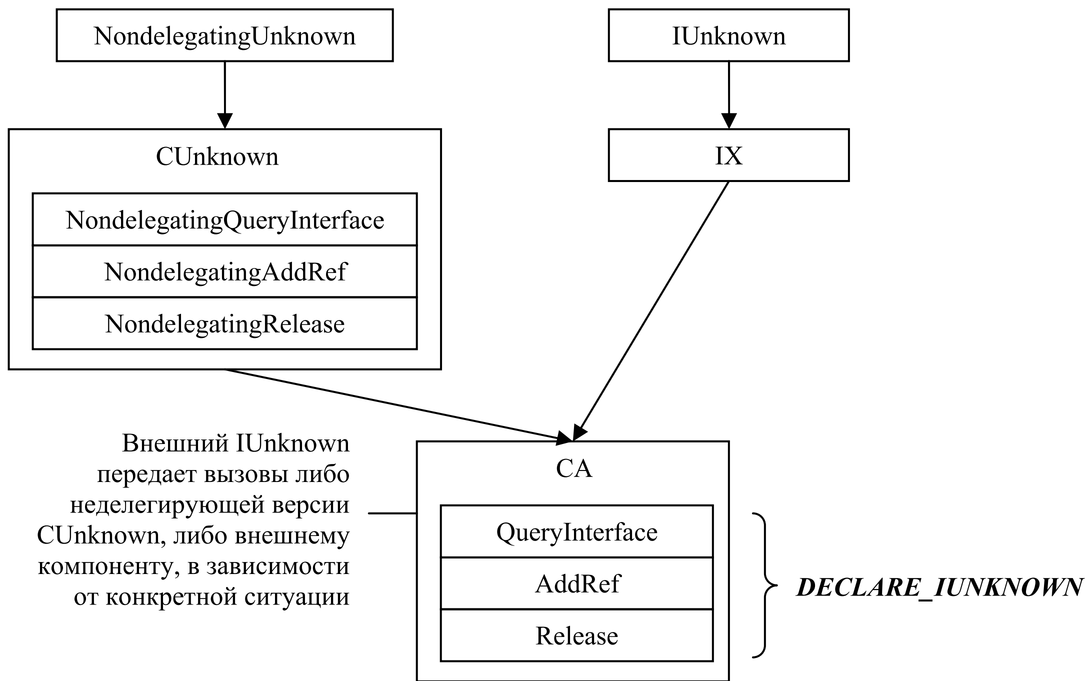


Рис. 9-3 *Наследуемый CUnknown не реализует интерфейс IUnknown, наследуемый от IX. Вместо этого CUnknown реализует INondelegatingUnknown. Сам компонент реализует IUnknown*

Функция *GetOuterUnknown*

Представленная в листинге 9-3 реализация DECLARE_IUNKNOWN использует функцию *CUnknown::GetOuterUnknown*, которая возвращает указатель на *IUnknown*. Если компонент наследует *CUnknown* и не агрегируется, *GetOuterUnknown* возвращает указатель на интерфейс *INondelegatingUnknown* этого компонента.

Если компонент агрегируется, эта функция возвращает указатель на интерфейс *IUnknown* внешнего компонента. Всякий раз, когда компоненту нужен указатель *IUnknown*, он использует *GetOuterUnknown*. Примечание: функция *GetOuterUnknown* не вызывает для возвращаемого ею указателя *AddRef*, так как в большинстве случаев число ссылок на этот интерфейс не нужно увеличивать.

Конструктор *CUnknown*

Что касается внешнего *IUnknown*, конструктор *CUnknown* принимает указатель на него в качестве параметра и сохраняет для последующего использования функцией *GetOuterUnknown*. Конструкторы классов, производных от *CUnknown*, должны также принимать указатель на внешний *IUnknown* и передавать его конструктору *CUnknown*.

Функции *Init* и *FinalRelease*

CUnknown поддерживает две виртуальные функции, которые помогают производным классам управлять внутренними компонентами. Чтобы создать компоненты для агрегирования или включения, производные классы переопределяют функцию *Init*. *CUnknown::Init* вызывается *CFactory::CreateInstance* сразу после создания компонента. *CUnknown::FinalRelease* вызывается из *CUnknown::NondelegatingRelease* непосредственно перед удалением компонента. Это дает компоненту возможность освободить имеющиеся у него указатели внутренних компонентов. *CUnknown::FinalRelease* увеличивает счетчик

ссылок во избежание рекурсивных вызовов *Release*, когда компонент освобождает интерфейсы содержащихся в нем компонентов.

Скоро Вы увидите, как просто реализовать компонент с помощью *CUnknown*. Но сначала давайте рассмотрим класс *CFactory*, который упрощает регистрациюи создание компонентов.

Базовый класс *CFactory*

Когда компонент реализован, необходимо создать для него фабрику класса. В предыдущих главах мы реализовывали фабрику класса заново для каждого компонента. В этой главе фабрика класса будет заранее реализована классом C++ *CFactory*. *CFactory* не только реализует интерфейс *IClassFactory*, он также предоставляет код, который можно использовать при реализации точек входа DLL, таких как *DllGetClassObject*. Кроме того, *CFactory* поддерживает размещение нескольких компонентов в одной DLL. Все такие компоненты будут совместно использовать одну реализацию *IClassFactory*. (Мы уже кратко рассматривали этот вариант в гл. 8.) *CFactory* можно использовать с любым компонентом, который удовлетворяет следующим трем требованиям:

- Компонент должен реализовывать функцию создания, имеющую прототип:

  ```
  HRESULT CreateFunction(IUnknown* pUnknownOuter, CUnknown**
  ppNewComponent)
  ```

  Универсальная реализация *IClassFactory::CreateInstance* должна иметь универсальный способ создания компонентов.
- Компонент должен наследовать *CUnknown*. Реализация *IClassFactory::CreateInstance* в *CFactory* после создания компонента вызывает *CUnknown::Init*. Компонент может переопределить этот метод для выполнения дополнительной инициализации, например, чтобы создать другой компонент для включения или агрегирования.
- Компонент должен заполнить структуру *CfactoryData* и поместить ее в глобальный массив *g_FactoryDataArray*.

Замечу, что данные требования не имеют никакого отношения к COM. Они обусловлены выбранным мною способом реализации *CFactory*. Давайте более подробно рассмотрим первое и последнее требования.

Прототип функции создания

CFactory нужен некий способ создания компонента. Она может использовать для этого любую функцию с таким прототипом:

```
HRESULT CreateFunction(IUnknown* pUnknownOuter, CUnknown** ppNewComponent)
```

Обычно такая функция создания вызывает конструктор компонента и затем возвращает указатель на новый компонент через выходной параметр *ppNewComponent*. Если первый параметр — *pUnknownOuter* — не равен NULL, то компонент агрегируется.
Я предпочитаю делать эту функцию статическим членом класса, реализующего компонент. Благодаря этому она оказывается в том же пространстве имен (name space), что и компонент. За исключением прототипа, эта функция может быть реализована аналогично функции *CreateInstance* предыдущих глав.

Данные компонента для *CFactory*

CFactory необходимо знать, какие компоненты он может создавать. Глобальный массив с именем *g_FactoryDataArray* содержит информацию об этих компонентах. Элементы массива *g_FactoryDataArray* — это классы *CfactoryData*. *CfactoryData* объявлен так:

```
typedef HRESULT (*FPCREATEINSTANCE)(IUnknown*, CUnknown**);
class CFactoryData
{
public:
  // Идентификатор класса компонента
  const CLSID* m_pCLSID;

  // Указатель на функцию, создающую компонент
  FPCREATEINSTANCE CreateInstance;

  // Имя компонента для регистрации в Реестре
  const char* m_RegistryName;

  // ProgID
  const char* m_szProgID;

  // Не зависящий от версии ProgID
  const char* m_szVerIndProgID;

  // Вспомогательная функция для поиска по идентификатору класса
  BOOL IsClassID(const CLSID& clsid) const
  {
    return (*m_pCLSID == clsid);
  }
};
```

CFactoryData имеет пять полей: идентификатор класса компонента, указатель функции создания компонента, дружественное имя для записи в Реестр Windows, ProgID и независимый от версии ProgID. В классе также имеется вспомогательная функция для поиска по идентификатору класса. Как видно из листинга 10.4, где показан файл SERVER.CPP, заполнить структуру *CFactoryData* нетрудно.

SERVER.CPP

```
#include "CFactory.h"
#include "Iface.h"
#include "Cmpnt1.h"
#include "Cmpnt2.h"
#include "Cmpnt3.h"

///////////////////////////////////////////////////////////
//
// Server.cpp
//
// Код сервера компонента.
// FactoryDataArray содержит компоненты, которые могут
// обслуживаться данным сервером.
//
// Каждый производный от CUnknown компонент определяет
// для своего создания статическую функцию со следующим прототипом:
// HRESULT CreateInstance(IUnknown* pUnknownOuter,
// CUnknown** ppNewComponent);
// Эта функция вызывается при создании компонента.
//
//
```

```
// Следующий далее массив содержит данные, используемые CFactory
// для создания компонентов. Каждый элемент массива содержит CLSID,
// указатель на функцию создания и имя компонента для помещения в
// Реестр.
//

CFactoryData g_FactoryDataArray[] =
{
  {&CLSID_Component1, CA::CreateInstance,
            "Inside COM, Chapter 9 Example, Component 1", // Дружественное имя
            "InsideCOM.Chap09.Cmpnt1.1", // ProgID
            "InsideCOM.Chap09.Cmpnt1"}, // Не зависящий от версии
  // ProgID
  {&CLSID_Component2, CB::CreateInstance,
            "Inside COM, Chapter 9 Example, Component 2",
            "InsideCOM.Chap09.Cmpnt2.1",
            "InsideCOM.Chap09.Cmpnt2"},
  {&CLSID_Component3, CC::CreateInstance,
            "Inside COM, Chapter 9 Example, Component 3",
            "InsideCOM.Chap09.Cmpnt3.1",
            "InsideCOM.Chap09.Cmpnt3"}
};

int g_cFactoryDataEntries = sizeof(g_FactoryDataArray) /
sizeof(CFactoryData);
```

Листинг 10.4 *Для того, чтобы использовать фабрику класса, Вы должны создать массив g_FactoryDataArray и поместить в него структуру CFactoryData для каждого компонента*

В листинге 10.4 элементы *g_FactoryDataArray* инициализированы информацией о трех компонентах, которые будет обслуживать данная DLL.

CFactory использует массив *g_FactoryDataArray*, чтобы определить, какие компоненты он может создавать. Если компонент присутствует в этом массиве, *CFactory* может его создать. *CFactory* получает из этого массива указатель на функцию создания компонента. Кроме того, *CFactory* использует информацию из *CFactoryData* для регистрации компонента. На рис. 10.4 показана структура сервера компонентов в процессе, реализованного с помощью *CFactory*.

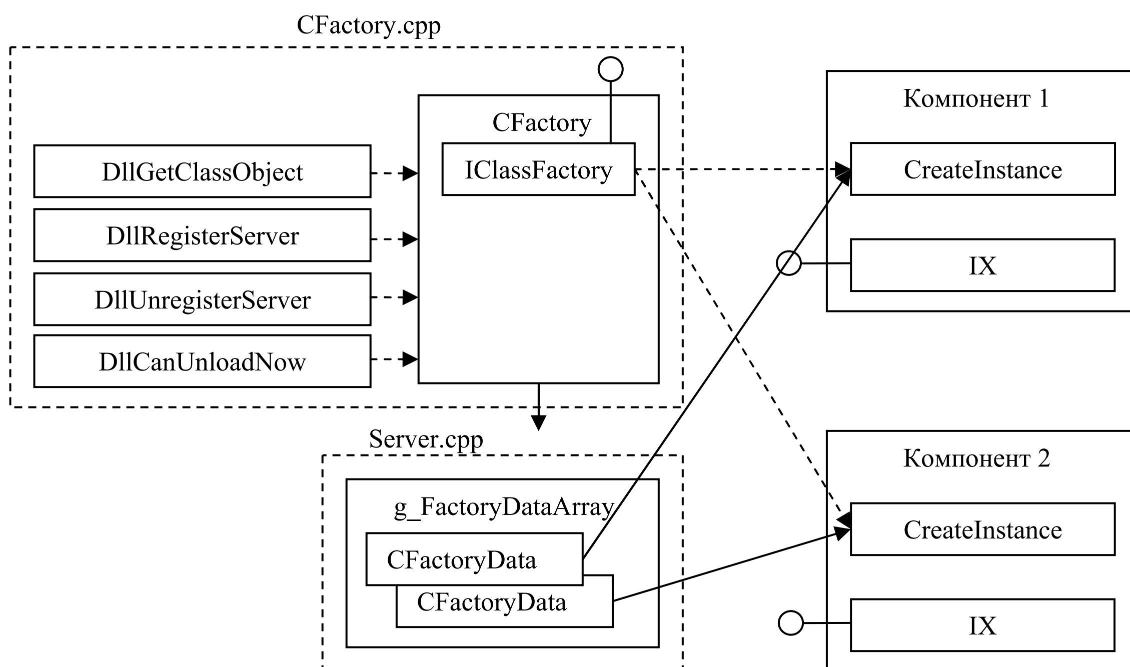


Рис. 10.4 *Организация CFactoryData*

Файл CFACTORY.H реализует различные точки входа DLL при помощи вызова функций *CFactory*. *DllGetClassObject* вызывает статическую функцию *Factory::GetClassObject*, и та обыскивает в глобальном массиве *g_FactoryDataArray* структуру *CfactoryData*, соответствующую компоненту, которого хочет создать клиент. Массив *g_FactoryDataArray* определен в SERVER.CPP и содержит информацию о всех компонентах, поддерживаемых данной DLL. *CFactory::GetClassObject* создает для компонента фабрику класса и передает последней структуру *CfactoryData*, соответствующую компоненту.

После создания компонента *CFactory* вызывается реализация *IClassFactory::CreateInstance*, которая для создания компонента использует указатель на функцию создания экземпляра, хранящийся в *CfactoryData*. Код *CFactory::GetClassObject* и *CFactory::CreateInstance* представлен в листингах 10.5 и 10.6.

Реализация *GetClassObject* в CFACTORY.CPP

```
///////////////////////////////////////////////////////////
//
// GetClassObject
// - Создание фабрики класса по заданному CLSID
//
HRESULT CFactory::GetClassObject(const CLSID& clsid, const IID& iid, void**
ppv)
{
  if ((iid != IID_IUnknown) && (iid != IID_IClassFactory))
  {
    return E_NOINTERFACE;
  }

  // Поиск идентификатора класса в массиве
  for (int i = 0; i < g_cFactoryDataEntries; i++)
  {
    const CFactoryData* pData = &g_FactoryDataArray[i];
    if (pData->IsClassID(clsid))
    {
      // Идентификатор класса найден в массиве компонентов,
```

```
      // которые мы можем создать. Поэтому создадим фабрику
      // класса для данного компонента. Чтобы задать фабрике
      // класса тип компонентов, которые она должна создавать,
      // ей передается структура CFactoryData
      *ppv = (IUnknown*) new CFactory(pData);
      if (*ppv == NULL)
      {
        return E_OUTOFMEMORY;
      }
      return NOERROR;
    }
  }
  return CLASS_E_CLASSNOTAVAILABLE;
}
```

Листинг 10.5 *Реализация CFactory::GetClassObject*

Реализация *CreateInstance* в CFACTORY.CPP

```
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
const IID& iid,
void** ppv)
{
  // При агрегировании IID должен быть IID_IUnknown
  if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
  {
    return CLASS_E_NOAGGREGATION;
  }

  // Создать компонент
  CUnknown* pNewComponent;
  HRESULT hr = m_pFactoryData->CreateInstance(pUnknownOuter, &pNewComponent);
  if (FAILED(hr))
  {
    return hr;
  }

  // Initialize the component.
  hr = pNewComponent->Init();
  if (FAILED(hr))
  {
    // Ошибка инициализации. Удалить компонент
    pNewComponent->NondelegatingRelease();
    return hr;
  }

  // Получить запрошенный интерфейс
  hr = pNewComponent->NondelegatingQueryInterface(iid, ppv);
  // Освободить ссылку, удерживаемую фабрикой класса
  pNewComponent->NondelegatingRelease();
  return hr;
}
```

Листинг 10.6 *Реализация CFactory::CreateInstance*

Вот и все тонкости создания компонентов с помощью *CFactory*. Реализуйте компонент и поместите его данные в структуру — это все!

Использование *CUnknown* и *CFactory*

Я очень рад, что теперь мы сможем повторно использовать реализацию интерфейс *IUnknown* и фабрики класса. Вам наверняка уже надоел один и тот же код *QueryInterface*, *AddRef* и *Release*. Я тоже устал от него. Отныне наши компоненты не будут реализовывать *AddRef* и *Release*, а будут лишь добавлять поддержку нужных интерфейсов в *QueryInterface*. Мы также сможем использовать простую функцию создания, а не писать заново целую фабрику класса. Наши новые клиенты будут похожи на клиент, представленный в листинге 10.7.

CMPNT2.H

```
//
// Cmpnt2.h - Компонент 2
//

#include "Iface.h"
#include "CUnknown.h" // Базовый класс для IUnknown

///////////////////////////////////////////////////////////
//
// Компонент B
//
class CB : public CUnknown, public IY
{
public:
  // Создание
  static HRESULT CreateInstance(IUnknown* pUnknownOuter,
  CUnknown** ppNewComponent);
private:
  // Объявление делегирующего IUnknown
  DECLARE_IUNKNOWN

  // Неделегирующий IUnknown
  virtual HRESULT __stdcall
  NondelegatingQueryInterface(const IID& iid, void** ppv);

  // Интерфейс IY
  virtual void __stdcall Fy();

  // Инициализация
  virtual HRESULT Init();

  // Очистка
  virtual void FinalRelease();

  // Конструктор
  CB(IUnknown* pUnknownOuter);

  // Деструктор
  ~CB();

  // Указатель на внутренний агрегируемый объект
  IUnknown* m_pUnknownInner;
  // Указатель на интерфейс IZ, поддерживаемый внутренним компонентом
  IZ* m_pIZ;
};
```

Листинг 10.7 *Компонент, использующий IUnknown, реализованный в CUnknown*

В листинге 10.7 представлен заголовочный файл для Компонента 2 из примера этой главы. Код мы рассмотрим чуть позже. В этом примере Компонент 1 реализует интерфейс *IX* самостоятельно. Для того, чтобы предоставить интерфейсы *IY* и *IZ*, он агрегирует Компонент 2. Компонент 2 реализует *IY* и агрегирует Компонент 3, который, в свою очередь, реализует *IZ*. Таким образом, Компонент 2 — одновременно и агрегируемый, и агрегирующий. Посмотрим листинг 10.7 от начала до конца. Я отмечу все интересные моменты, а затем мы рассмотрим их подробно.

Компонент наследует *CUnknown*, который предоставляет реализацию *IUnknown*. Мы объявляем статическую функцию, которую *CFactory* будет использовать для создания компонента. Имя этой функции для *CFactory* не имеет значения, поэтому можно назвать ее как угодно.

Далее мы реализуем делегирующий *IUnknown* с помощью макроса DECLARE_IUNKNOWN. DECLARE_IUNKNOWN реализует делегирующий *IUnknown*, а *CUnknown* — неделегирующий. Хотя *CUnknown* полностью реализует *AddRef* и *Release*, он не может предоставить полной реализации *QueryInterface*, так как ему неизвестно, какие интерфейсы поддерживает наш компонент. Поэтому компонент реализует *NondelegatingQueryInterface* для обработки запросов на его собственные интерфейсы.
Производные классы переопределяют *Init* для создания внутренних компонентов при агрегировании или включении. *CUnknown::NondelegatingRelease* вызывает *FinalRelease* непосредственно перед тем, как удалить объект. Последнюю переопределяют те компоненты, которым необходимо освободить указатели на внутренние компоненты. *CUnknown::FinalRelease* увеличивает счетчик ссылок, чтобы предотвратить рекурсивную ликвидацию компонента.

Теперь рассмотрим различные аспекты Компонента 2, код которого представлен в листинге 10.8.

CMPNT2.CPP

```
//
// Cmpnt2.cpp - Компонент 2
//

#include <objbase.h>

#include "Iface.h"
#include "Util.h"
#include "CUnknown.h" // Базовый класс для IUnknown
#include "Cmpnt2.h"

static inline void trace(char* msg)
{ Util::Trace("Компонент 2", msg, S_OK); }

static inline void trace(char* msg, HRESULT hr)
{ Util::Trace("Компонент 2", msg, hr); }

///////////////////////////////////////////////////////////
//
// Реализация интерфейса IY
//
void __stdcall CB::Fy()
{
  trace("Fy");
}
```

```
//
// Конструктор
//

CB::CB(IUnknown* pUnknownOuter)
: CUnknown(pUnknownOuter), m_pUnknownInner(NULL), m_pIZ(NULL)
{
  // Пустой
}

//
// Деструктор
//
CB::~CB()
{
  trace("Самоликвидация");
}

//
// Реализация NondelegatingQueryInterface
//
HRESULT __stdcall CB::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IY)
  {
    return FinishQI(static_cast<IY*>(this), ppv);
  }
  else if (iid == IID_IZ)
  {
    return m_pUnknownInner->QueryInterface(iid, ppv);
  }
  else
  {
    return CUnknown::NondelegatingQueryInterface(iid, ppv);
  }
}

//
// Инициализировать компонент и создать внутренний компонент
//
HRESULT CB::Init()
{
  trace("Создание агрегируемого Компонента 3");
  HRESULT hr = CoCreateInstance(CLSID_Component3,
                      GetOuterUnknown(),
                      CLSCTX_INPROC_SERVER,
                      IID_IUnknown,
                      (void**)&m_pUnknownInner);

  if (FAILED(hr))
  {
    trace("Не могу создать внутренний компонент", hr);
    return E_FAIL;
  }

  trace("Получить указатель на интерфейс IZ для использования в дальнейшем");
  hr = m_pUnknownInner->QueryInterface(IID_IZ, (void**)&m_pIZ);
  if (FAILED(hr))
  {
    trace("Внутренний компонент не поддерживает IZ", hr);
    m_pUnknownInner->Release();
    m_pUnknownInner = NULL;
    return E_FAIL;
  }
```

```
  // Компенсировать увеличение счетчика ссылок из-за вызова QI
  trace("Указатель на интерфейс IZ получен. Освободить ссылку.");
  GetOuterUnknown()->Release();
  return S_OK;
}

//
// FinalRelease – Вызывается из Release перед удаление компонента
//

void CB::FinalRelease()
{
  // Вызов базового класса для увеличения m_cRef и предотвращения рекурсии
  CUnknown::FinalRelease();

  // Учесть GetOuterUnknown()->Release в методе Init
  GetOuterUnknown()->AddRef();

  // Корректно освободить указатель, так как подсчет ссылок
  // может вестись поинтерфейсно
  m_pIZ->Release();

  // Освободить внутренний компонент
  // (Теперь мы можем это сделать, так как освободили его интерфейсы)
  if (m_pUnknownInner != NULL)
  {
    m_pUnknownInner->Release();
  }
}

///////////////////////////////////////////////////////////
//
// Функция создания для CFactory
//
HRESULT CB::CreateInstance(IUnknown* pUnknownOuter, CUnknown**
ppNewComponent)
{
  *ppNewComponent = new CB(pUnknownOuter);
  return S_OK;
}
```

Листинг 10.8 *Реализация компонента, использующего CUnknown и CFactory*

NondelegatingQueryInterface

Вероятно, самая интересная часть компонента — *NondelegatingQueryInterface*. Мы реализуем ее почти так же, как *QueryInterface* в предыдущих главах. Обратите, однако, внимание на два отличия. Во-первых, мы используем функцию *FinishQI*, причем делаем это лишь для удобства; мы не обязаны ее использовать. *FinishQI* лишь несколько облегчает реализацию *NondeletgatingQueryInterface* в производных классах. Код этой функции показан ниже:

```
HRESULT CUnknown::FinishQI(IUnknown* pI, void** ppv)
{
  ppv = pI;
 I->AddRef();
 turn S_OK;
}
```

Второе отличие в том, что нам нет необходимости обрабатывать запрос на *IUnknown*. Базовый класс выполняет обработку для *IUnknown* и всех интерфейсов, о которых мы не знаем:

```
HRESULT __stdcall CUnknown::NondelegatingQueryInterface(const IID& iid,
void** ppv)
{
  // CUnknown поддерживает только IUnknown
  if (iid == IID_IUnknown)
  {
    return FinishQI(reinterpret_cast<IUnknown*>
    (static_cast<INondelegatingUnknown*>(this)), ppv);
  }
  else
  {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
}
```

Все вместе, шаг за шагом

Приведенный выше код показывает, как легко писать компоненты с помощью *CUnknown* и *CFactory*. Давайте рассмотрим всю процедуру целиком. Далее приводится последовательность шагов создания компонента, его фабрики класса и DLL, в которой он будет находиться:

1. Напишите класс, реализующий компонент.
   - Базовым классом компонента должен быть либо *CUnknown*, либо другой класс, производный от него.
   - При помощи макроса DECLARE_IUNKNOWN реализуйте делегирующий *IUnknown*.
   - Инициализируйте *CUnknown* в конструкторе своего класса.
   - Реализуйте *NondelegatingQueryInterface*, добавив интерфейсы, которые поддерживает Ваш класс, но не поддерживает базовый класс. Вызовите базовый класс для тех интерфейсов, которые не обрабатываются в Вашем классе.
   - Если Ваш компонент требует дополнительной инициализации после конструктора, реализуйте функцию *Init*. Создайте включаемые и агрегируемые компоненты, если это необходимо.
   - Если после освобождения, но перед удалением компонент должен выполнить какую-либо очистку, реализуйте *FinalRelease*. Освободите указатели на включаемые и агрегируемые компоненты.
   - Реализуйте для своего компонента статическую функцию *CreateInstance*.
   - Реализуйте поддерживаемые компонентом интерфейсы.

2. Повторите шаг 1 для каждого из компонентов, которые Вы хотите поместить в данную DLL.

3. Напишите фабрику класса.
   - Создайте файл для определения глобального массива *CfactoryData* — *g_FactoryDataArray*.
   - Определите *g_FactoryDataArray* и поместите в него информацию о всех компонентах, обслуживаемых этой DLL.

- Определите *g_cFactoryDataEntries*, которая должна содержать число компонентов в массиве *g_FactoryDataArray*.

4. Создайте DEF файл с описанием точек входа в DLL.

5. Скомпилируйте и скомпонуйте свою программу вместе с файлами CUNKNOWN.CPP и CFACTORY.CPP.

6. Пошлите мне в знак благодарности открытку с изображением водопада или речного порога. Весь процесс очень прост. Я могу создать новый компонент меньше, чем за пять минут.

Резюме

При использовании класса smart-указателей, скрывающих подсчет ссылок, работа с компонентами COM становится похожей на работу с классами C++. Кроме того, применение smart-указателей помогает уменьшить число ошибок, поскольку обеспечивает безопасное в смысле приведения типов получение указателей на интерфейсы. Многие классы smart-указателей на интерфейсы переопределяют *operator=*, чтобы вызывать *QueryInterface* при присваивании указателя на интерфейс одного типа указателю на интерфейс другого типа. Если smart-указатели облегчают работу с объектами COM, то некоторые классы C++ делают ее максимально простой. Классы *CUnknown* и *CFactory* упрощают создание компонентов COM, предоставляя повторно применимые реализации *IUnknown* и *IClassFactory*.

Учитывая всеобщее стремление до предела все упростить, я удивлюсь, если после этой главы Вам не станет легче дышать. Да, чуть не забыл — *есть* компания, производящая устройство для облегчения дыхания, вставляемое в нос. Некоторые велосипедисты-профессионалы его используют. Я полагаю, *немного* помощи не повредит, когда Вы делаете что-то новое.

11. Серверы в EXE

В последний раз я был в Берлине еще до падения стены. Когда, покидая американский сектор у пропускного поста «Чекпойнт Чарли», я въезжал в Восточный Берлин, не возникало сомнений, что здесь проходит граница. Колючая проволока, автоматчики и минные поля делали ее весьма отчетливой. Но и за оборонительной линией отличия были очевидны: с восточной стороны от стены двухлитровые малолитражки изрыгали густой дым, а возле магазинов стояли длинные очереди.

Изменения ждут нас при всяком переходе границы, неважно, сколь мало отличается одна сторона от другой. Эта глава посвящена пересечению границ — главным образом, границ между разными процессами. Мы рассмотрим также пересечение границ между машинами.

Почему нам нужно выходить за границу процесса? Потому, что в некоторых случаях предпочтительнее реализовать компонент в EXE, а не в DLL. Одной из причин может стать то, что Ваше приложение уже реализовано в EXE. После небольшой доработки можно сделать доступными сервисы приложения, так что клиенты смогут автоматизировать его использование.

Если компонент и клиент находятся в разных EXE, они будут расположены и в отдельных процессах, поскольку для каждого EXE-модуля создается свой процесс. При передаче информации между такими компонентом и клиентом необходимо пересечь границу между процессами. По счастью, при этом нет нужды изменять код компонента, хотя некоторые изменения в класс *CFactory*, представленный в предыдущей главе, внести все же придется. Однако, прежде чем перейти к реализации, следует рассмотреть проблемы и решения, связанные с обращением к интерфейсам COM через границы процессов.

Разные процессы

Каждый модуль EXE исполняется в отдельном процессе. У каждого процесса есть свое адресное пространство. Логический адрес 0x0000ABBA в двух разных процессах ссылается на два разных места в физической памяти. Если один процесс передаст этот адрес другому, второй будет работать не с тем участком памяти, который предполагался первым процессом (рис. 11.1).

В то время как каждому EXE-модулю соответствует свой процесс, DLL проецируется в процесс того EXE, с которым они скомпонованы. По этой причине DLL называют серверами *внутри процесса (in process)*, а EXE — серверами *вне процесса (out of process)*. Иногда EXE также называют *локальными серверами*, чтобы отличить их от другого вида серверов вне процесса — *удаленных серверов*. Удаленный сервер — это сервер вне процесса, работающий на другой машине.

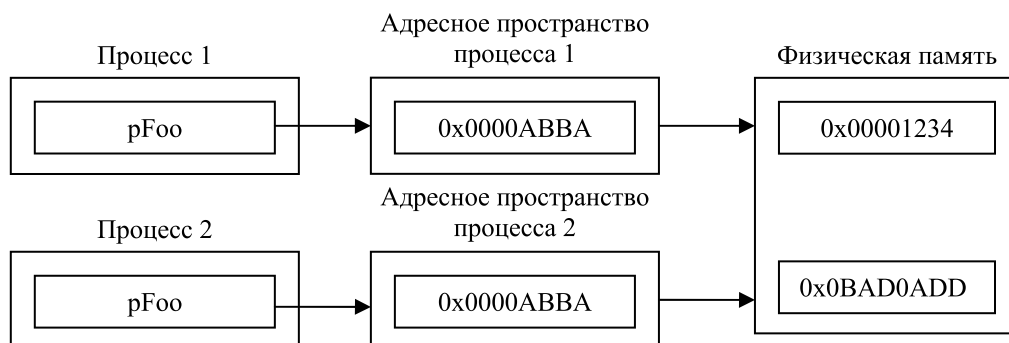


Рис. 11.1 *Один и тот же адрес в двух разных процессах ссылается на два разных участка физической памяти*

В гл. 6 мы говорили, как важно то, что компонент и клиент используют общее адресное пространство. Компонент передает клиенту интерфейс. Интерфейс — это, по существу, массив указателей функций. Клиент должен иметь доступ к памяти, занимаемой интерфейсом. Если компонент находится в DLL, то доступ осуществляется легко: и компонент, и клиент находятся в одном адресном пространстве. Но если компонент и клиент находятся в разных адресных пространствах, то у клиента нет доступа к памяти процесса компонента. Если у клиента нет даже доступа к памяти, связанной с интерфейсом, то он не сможет вызывать и функции этого интерфейса. В такой ситуации наши интерфейсы стали бы совершенно бесполезны.

Для того, чтобы с интерфейсом можно было работать через границы процесса, необходимо следующее:

- Процесс должен иметь возможность вызвать функцию в другом процессе.
- Процесс должен иметь возможность передавать другому процессу данные.
- Клиент не должен беспокоиться о том, является ли компонент сервером внутри или вне процесса.

Локальный вызов процедуры

Есть много методов межпроцессной коммуникации, включая DDE, именованные каналы и разделяемую память. Однако COM использует *локальный вызов процедуры (local procedure call, LPC)*. LPC — это средство связи между разными процессамина одной и той же машине. LPC представляет собой специализированное средство связи между разными процессами в пределах одной машины, построенное на основе *удаленного вызова процедуры (remote procedure call, RPC)* (см. рис. 11.2).

Стандарт RPC определен OSF (Open Software Foundation) в спецификации DCE (Distributed Computing Environment) RPC. RPC обеспечивает коммуникацию между процессами на разных машинах с помощью разнообразных сетевых протоколов. Распределенная модель COM (DCOM), которую мы будем рассматривать далее в этой главе, использует RPC для связи по сети.

Как работает RPC? Как по волшебству. На самом деле волшебства, конечно, нет, но есть кое-что, что лишь немногим хуже, — реализация вызовов операционной системой. Операционной системе известны физические адреса, соответствующие логическому адресному пространству каждого процесса; следовательно, операционная система может вызывать функции внутри любого процесса.

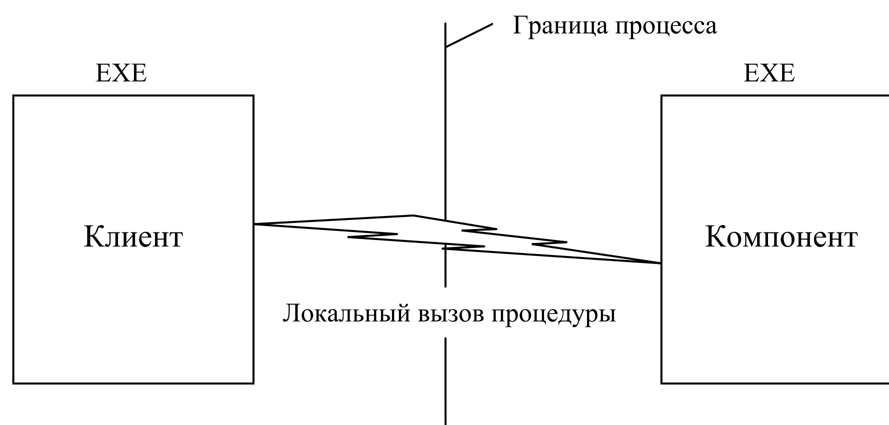


Рис. 11.2 *Клиент в EXE использует механизм Win32 LPC для вызова функций компонента, реализованного в другом EXE*

Маршалинг

Вызвать функцию EXE — это только полдела. Нам по-прежнему необходимо передать параметры функции из адресного пространства клиента в адресное пространство компонента. Этот процесс называется *маршалингом (marshaling)*. В соответствии с моим словарем, глагол marshal значит «располагать, размещать или устанавливать в определенном порядке». Это слово должно быть в нашем ближайшем диктанте.

Если оба процесса находятся на одной машине, маршалинг выполняется просто. Данные одного процесса необходимо скопировать в адресное пространство другого процесса. Если процессы находятся на разных машинах, то данные необходимо преобразовать в стандартный формат, учитывающий межмашинные различия, например, порядок следования байтов в слове.

Механизм LPC способен скопировать данные из одного процесса в другой. Но для выборки параметров и посылки их в другой процесс ему требуется больше информации, чем содержится в заголовочном файле C++.

Например, указатели на структуры следует обрабатывать иначе, чем целые числа. Маршалинг указателя включает в себя копирование в другойпроцесс структуры, на которую указатель ссылается. Однако, если указатель — это указатель на интерфейс, то область памяти, на которую он ссылается, копироваться не должна. Как видите, для выполнения маршалинга нужно сделать больше, чем просто вызвать *memcpy*.

Для маршалинга компонента предназначен интерфейс *IMarshal*. В процессе создания компонента СОМ запрашивает у него этот интерфейс. Затем СОМ вызывает функции-члены этого интерфейса для маршалинга и демаршалинга параметров до и после вызова функций. Библиотека СОМ реализует стандартную версию *IMarshal*, которая работает для большинства интерфейсов. Основной причиной создания собственной версии *IMarshal* является стремление повысить производительность. Подробно маршалинг описан в книге Крейга Брокшмидта *Inside OLE*.

DLL заместителя/заглушки

Неужели я потратил девять глав на обсуждение того, как вызывать компоненты СОМ через интерфейсы, чтобы в десятой начать вызывать их посредством LPC? С самого начала мы стремились унифицировать работу клиента с компонентами — внутри процесса, вне процесса и удаленными. Очевидно, что мы на достигли цели, если клиенту нужно заботиться о LPC. СОМ решает проблему просто.

Хотя большинство разработчиков для Windows этого и не знают, они используют LPC практически при любом вызове функции Win32. Вызов функции Win32 вызывает функцию DLL, которая через LPC вызывает код Windows, фактически выполняющий работу. Такая процедура изолирует Вашу программу, находящуюся в своем процессе, от кода Windows. Так как у разных процессов разные адресные пространства, Ваша программа не сможет разрушить операционную систему.

СОМ использует весьма похожую структуру. Клиент взаимодействует с DLL, которая изображает собой компонент. Эта DLL выполняет за клиента маршалинг и вызовы LPC. В СОМ такой компонент называется *заместителем (proxy)*.

В терминах СОМ, заместитель — это компонент, который действует как другой компонент. Заместители должны находиться в DLL, так как им необходим доступ к адресному пространству клиента для маршалинга данных, передаваемых ему функциями интерфейса. Но маршалинг — это лишь половина дела; компоненту еще требуется DLL, называемая *заглушкой (stub)*, для демаршалинга данных, переданных клиентом. Заглушка выполняет также маршалинг данных, возвращаемых компонентом обратно клиенту (рис. 11.3).

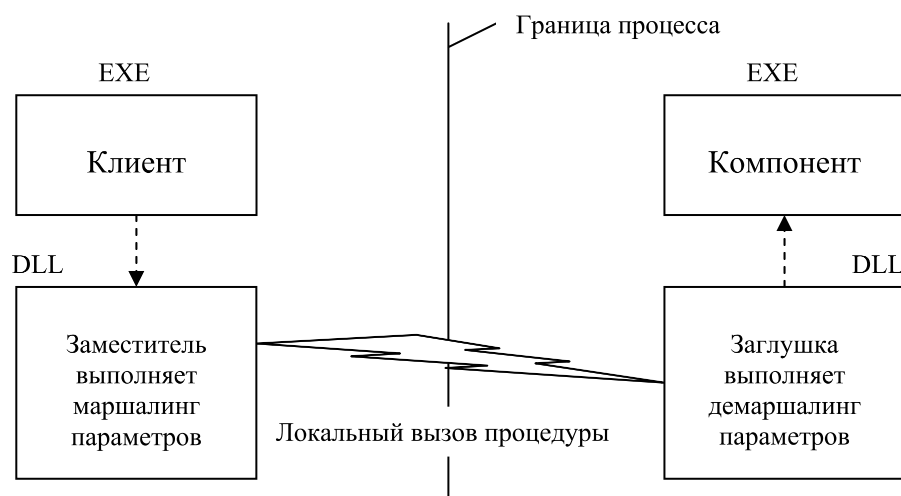


Рис. 11.3 *Клиент работает с DLL заместителя. Заместитель выполняет маршалинг параметров фунции и вызывает DLL заглушки с помощью LPC. DLL заглушки выполняет демаршалинг параметров и вызывает соответствующую функцию компонента, передавая ей параметры.*

Данный процесс изображен на рис. 11.3 весьма упрощенно. Однако рисунок дает представление о том, как много нужно кода, чтобы все это работало.

Забудьте об этом! Слишком много кода!

Итак, для размещения компонента в EXE нужно написать заместитель и заглушку. Кроме того, нужно разбираться в LPC, чтобы реализовать вызовы через границы процессов. Вдобавок нужно реализовать *Imarshal* для маршалинга данных от клиента компоненту и обратно. Кажется, для меня это слишком много… я лучше проведу это время, бросая камешки в воду… К счастью, всю работу нам делать не нужно — по крайней мере, в большинстве случаев.

Введение в IDL/MIDL

Вероятно, Вам интереснее писать хорошие компоненты, чем кучу кода, предназначенного лишь для общения компонентов друг с другом. Я лучше займусь составлением программы OpenGL, чем кодом пересылки данных между двумя процессами. К счастью, мы не обязаны писать этот код сами. Задав описание интерфейса на языке *IDL (Interface Definition Language)*, мы можем использовать компилятор *MIDL* для автоматической генерации DLL заместителя/заглушки.

Конечно, если Вы хотите сделать все сами, — пожалуйста. Одна из привлекательных черт COM в том, что эта модель предоставляет множество реализаций по умолчанию, но позволяет, если нужно, создать и свой собственный интерфейс. Но для большинства интерфейсов специализированный маршалинг не нужен, и большинству компонентов не требуются самодельные заместители и заглушки. Иначе говоря, будем использовать *простой* способ. Конечно, «простой» — понятие относительное, и от нас по-прежнему потребуются определенные усилия.

Язык IDL, так же как UUID и спецификация RPC, был заимствован из OSF DCE. Его синтаксис похож на C и C++, и он обладает богатыми возможностями описания интерфейсов и данных, используемых клиентом и компонентом. Хотя интерфейс COM использует только подмножество IDL, Microsoft потребовалось внести некоторые

нестандартные расширения для поддержки COM. Мы в Microsoft всегда считаем, что стандарт можно улучшить.

После составления описания интерфейсов и компонентов на IDL это описание обрабатывается компилятором MIDL (компилятор IDL фирмы Microsoft). Последний генерирует код на C для DLL заместителя и заглушки. Остается просто откомпилировать эти файлы и скомпоновать их, чтобы получить DLL, реализующую нужный Вам код заместителя и заглушки! Поверьте, это гораздо лучше, чем делать все вручную.

IDL

Хотя Вы и избавились от труда изучения LPC, Вам все равно необходимо научиться описывать интерфейсы на IDL. Это нетрудно, но может вызвать сильное разочарование — IDL непоследователен, документация плохая, хорошие примеры найти трудно, а сообщения об ошибках иногда загадочны. Мое любимое: «Попробуйте обойти ошибку» («Try to find a work around»). Поскольку IDL сохранит нам массу времени и усилий, мы не будем (долго) сетовать по этому поводу. Мой совет — потратить день и прочитать документацию IDL на компакт-диске MSDN. Эта документация суха и утомительна, но гораздо лучше прочитать ее заранее, чем откладывать до той ночи, к концу которой Вам надо будет описать интерфейс на IDL.

У Вас может возникнуть искушение не использовать IDL, а сделать все самостоятельно. Но в следующей главе мы будем использовать компилятор MIDL для создания библиотек типа. Вы можете самостоятельно создавать и библиотеки типа, но такой подход не имеет никаких реальных преимуществ. Коротко говоря, гораздо лучше потратить время на изучение IDL, который сразу дает заместитель, заглушку и библиотеку типа.

Примеры описаний интерфейсов на IDL

Давайте рассмотрим пример описания интерфейса на IDL. Ниже приведен фрагмент файла SERVER.IDL

```
import "unknwn.idl";
// Интерфейс IX
[
      object,
      uuid(32bb8323-b41b-11cf-a6bb-0080c7b2d682),
      helpstring("IX Interface"),
      pointer_default(unique)
]

interface IX : IUnknown
{
      HRESULT FxStringIn([in, string] wchar_t* szIn);
      HRESULT FxStringOut([out, string] wchar_t** szOut);
};
```

На C++ соответствующие функции выглядели бы так:

```
virtual HRESULT __stdcall FxStringIn(wchar_t* szIn);
virtual HRESULT __stdcall FxStringOut(wchar_t** szOut);
```

Как видите, синтаксис MIDL не очень отличается от синтаксиса C++. Самое очевидное отличие — информация в квадратных скобках ([]). Перед каждым интерфейсом идет список атрибутов, или заголовок интерфейса. В данном примере заголовок состоит из четырех записей. Ключевое слово *object* задает, что данный интерфейс является

интерфейсом COM. Это ключевое слово представляет собой расширение IDL от Microsoft. Второе ключевое слово — *uuid* — задает IID интерфейса. Третье ключевое слово используется для помещения строки подсказки в библиотеку типа. Подождите, мы еще рассмотрим библиотеки типа в следующей главе, так как они связаны напрямую с серверами вне процесса. Четвертое ключевое слово — *pointer_default* — выглядит незнакомо и мы сейчас о нем поговорим.

Ключевое слово *pointer_default*

Назначение IDL состоит лишь в том, чтобы предоставить достаточные сведения для маршалинга параметров функций. Для этого IDL нужна информация о том, как работать с некоторыми вещами. Ключевое слово *pointer_default* говорит компилятору MIDL, как работать с указателями, атрибуты которых не заданы явно.

Имеется три опции:

- *ref* — указатели рассматриваются как ссылки. Они всегда будут содержать допустимый адрес памяти и всегда могут быть разыменованы. Они не могут иметь значение NULL. Они указывают на одно и то же место в памяти как до, так и после вызова. Кроме того, внутри функции для них нельзя создавать синонимы (aliases).
- *unique* — эти указатели могут иметь значение NULL. Кроме того, их значение может изменяться при работе функции. Однако внутри функции для них нельзя создавать синонимы.
- *ptr* — эта опция указывает, что по умолчанию указатель эквивалентен указателю C. Указатель может иметь синоним, может иметь значение NULL и может изменяться.

MIDL использует эти значения для оптимизации генерируемого кода заместителя и заглушки.

Входные и выходные параметры в IDL

Для дальнейшей оптимизации заместителя и заглушки MIDL использует входные и выходные параметры (*in* и *out*). Если параметр помечен как входной, то MIDL знает, что этот параметр нужно передавать только от клиента компоненту. Заглушка не возвращает значение параметра обратно. Ключевое слово *out* говорит MIDL о том, что параметр используетсятолько для возврата данных от компонента клиенту. Заместителю не нужно выполнять маршалинг выходного параметра и пересылать его значение компоненту.
Параметры могут быть также помечены обоими ключевыми словами одновременно:

```
HRESULT foo([in] int x, [in, out] int* y, [out] int* z);
```

В предыдущем фрагменте *y* является как входным, так и выходным параметром. Компилятор MIDL требует, чтобы выходные параметры были указателями.

Строки в IDL

Чтобы выполнить маршалинг элемента данных, необходимо знать его длину, иначе его нельзя будет скопировать. Определить длину строки C++ легко — нужно найти завершающий ее символ с кодом 0. Если параметр функции имеет атрибут *string*, то MIDL знает, что этот параметр является строкой, и может указанным способом определить ее длину.

По стандартному соглашению COM использует для строки символы UNICODE, даже в таких системах, как Microsoft Windows 95, которые сами UNICODE не поддерживают. Именно по этой причине предыдущий параметр использует для строк тип *wchar_t*. Вместо *wchar_t* Вы также можете использовать OLECHAR или LPOLESTR, которые определены в заголовочных файлах COM.

HRESULT в IDL

Вы, вероятно, заметили, что обе функции представленного выше интерфейса *IX* возвращают HRESULT. MIDL требует, чтобы функции интерфейсов, помеченные как *object*, возвращали HRESULT. Основная причина этого — требование поддержки удаленных серверов. Если Вы подключаетесь к удаленному серверу, любая функция может потерпеть неудачу из-за ошибки сети. Следовательно, у каждой функции должен быть способ сигнализации о сетевых ошибках. Для этого проще всего потребовать, чтобы все функции возвращали HRESULT.

Именно поэтому большинство функций COM возвращает HRESULT. (Многие пишут для интерфейсов COM классы-оболочки, которые генерируют исключение, если метод возвращает код ошибки. На самом деле компилятор Microsoft Visual C++ версии 5.0 может импортировать библиотеку типа и автоматически сгенерировать для ее членов классы-оболочки, который будут генерировать исключения при получении ошибочных HRESULT.) Если функции нужно возвращать параметр, отличный от HRESULT, следует использовать выходной параметр. Функция *FxStringOut* использует такой параметр для возврата компонентом строки. Эта функция выделяет память для строки при помощи *CoTaskMemAlloc*. Клиент должен освободить эту память при помощи *CoTaskMemFree*. Следующий пример из файла CLIENT.CPP гл. 11 демонстрирует использование определенного выше интерфейса.

```
wchar_t* szOut = NULL;

HRESULT hr = pIX->FxStringIn(L"Это тест");
assert(SUCCEEDED(hr));

hr = pIX->FxStringOut(&szOut);
assert(SUCCEEDED(hr));

// Отобразить возвращенную строку
ostrstream sout;
sout << "FxStringOut возвратила строку: "
     << szOut // Использование переопределенного оператора << для типа
wchar_t
     << ends;
trace(sout.str());

// Удалить возвращенную строку
::CoTaskMemFree(szOut);
```

Для освобождения памяти используется *CoTaskMemFree*.

Ключевое слово *import* в IDL

Ключевое слово *import* используется для включения определений из других файлов IDL. UNKNWN.IDL содержит описание на IDL интерфейса *IUnknown*; *import* является аналогом команды препроцессора C++ *#include*, но с помощью *import* файл можно импортировать сколько угодно раз, не создавая проблем с повторными определениями.

Все стандартные интерфейсы COM и OLE (ActiveX) определены в файлах IDL (посмотрите в каталоге INCLUDE своего компилятора; просматривать файлы IDL для

стандартных интерфейсов OLE — хороший метод приобретения опыта описания интерфейсов).

Модификатор *size_is* в IDL

Теперь рассмотрим интерфейс, передающий массивы между клиентом и компонентом:

```
// Интерфейс IY
[
      object,
      uuid(32bb8324-b41b-11cf-a6bb-0080c7b2d682),
      helpstring("Интерфейс IY"),
      pointer_default(unique)
]

interface IY : IUnknown
{
      HRESULT FyCount([out] long* sizeArray);
      HRESULT FyArrayIn([in] long sizeIn,
      [in, size_is(sizeIn)] long arrayIn[]);
      HRESULT FyArrayOut([out, in] long* psizeInOut,
      [out, size_is(*psizeInOut)] long arrayOut[]);
};
```

У интерфейса тот же заголовок, что и раньше. Здесь нам интересен атрибут *size_is*. Одна из основных функций маршалинга состоит в копировании данных из одного места в другое. Следовательно, очень важно иметь информацию о размере данных. Если размер всегда фиксирован, проблем нет. Но если его можно определить только во время выполнения, задача становится несколько сложнее. Если мы передаем функции массив, каким образом заместитель определит его размер?

Именно для этого и предназначен атрибут *size_is*. Для функции *FyArrayIn* этот атрибут сообщает MIDL, что число элементов массива хранится в *sizeIn*. Аргументом *size_is* может быть только входной параметр или параметр типа вход-выход. *(in-out)*. Использование параметра вход-выход с атрибутом *size_is* демонстрирует другая функция интерфейса *IY — FyArrayOut*.

В качестве второго параметра клиент передает массив, для которого он уже выделил память. Количество элементов массива передается в первом параметре *psizeInOut*. Функция заполняет массив некоторыми данными. Затем она заносит в *psizeInOut* число элементов, которые фактически возвращает. По правде сказать, я не очень люблю параметры типа вход-выход и никогда не создаю интерфейсы, подобные только что приведенному. Вместо этого я определил бы отдельный выходной параметр (*out*) для возвращения компонентом числа заполненных элементов массива:

```
HRESULT FyArrayOut2([in] long sizeIn,
[out, size_is(sizeIn)] long arrayOut[],
[out] long* psizeOut);
```

Приведенный ниже код — фрагмент файла CLIENT.CPP из примера гл. 11. Здесь интерфейс *IY* сначала используется для передачи массива компоненту, а затем для возвращения его обратно.

```
// Послать массив компоненту
long arrayIn[] = { 22, 44, 206, 76, 300, 500 };
long sizeIn = sizeof(arrayIn) / sizeof(arrayIn[0]);
HRESULT hr = pIY->FyArrayIn(sizeIn, arrayIn);
assert(SUCCEEDED(hr));
```

```
// Получить массив от компонента обратно

// Получить размер массива
long sizeOut = 0;
hr = pIY->FyCount(&sizeOut);
assert(SUCCEEDED(hr));

// Выделить память для массива
long* arrayOut = new long[sizeOut];

// Получить массив
hr = pIY->FyArrayOut(&sizeOut, arrayOut);
assert(SUCCEEDED(hr));

// Отобразить массив, возращенный функцией
ostrstream sout;
sout << "FyArray вернула "
     << sizeOut
     << " элементов: ";

for (int i = 0; i < sizeOut, i++)
{
  sout << " " << arrayOut[i];
}
sout << "." << ends;
trace(sout.str());

// Очистка
delete [] arrayOut;
```

Технически, в соответствии со спецификацией COM, память для параметров типа *out* необходимо выделять с помощью *CoTaskMemAlloc*. Но многие интерфейсы COM эту функцию не используют. Наиболее близкий к *IY::FyArrayOut* пример — *IxxxxENUM::Next*, которая также не использует *CoTaskMemAlloc*. Самое странное в библиотеке COM то, что некоторые ее функции используют *CoTaskMemAlloc*, а некоторые нет. И по документации трудно отнести функцию к тому или другому классу: например, сравните документацию функций *StringFromCLSID* и *StringFromGUID2*. Какая из них требует освобождения памяти с помощью *CoTaskMemFree*? Если Вы не знаете — ответ в гл. 7.

Структуры в IDL

Я уверен, что Ваша программа передает функциям не только простые типы, но и структуры. Структуры в стиле C и C++ также можно определить в файле IDL и использовать как параметры функций. Например, представленный ниже интерфейс использует структуру, состоящую из трех полей:

```
// Структура для интерфейса IZ
typedef struct
{
  double x;
  double y;
  double z;
} Point3d;

// Интерфейс IZ
[
  object,
  uuid(32bb8325-b41b-11cf-a6bb-0080c7b2d682),
  helpstring("Интерфейс IZ"),
  pointer_default(unique)
```

```
]

interface IZ : IUnknown
{
  HRESULT FzStructIn([in] Point3d pt);
  HRESULT FzStructOut([in] Point3d* pt);
};
```

И здесь IDL очень похож на C++. Дело усложняется, если Вы передаете непростые структуры, содержащие указатели. MIDL необходимо точно знать, на что каждый из них указывает, чтобы выполнить маршалинг данных, на которые имеется ссылка. Поэтому не используйте в качестве типа параметра *void**. Если Вам нужно передать абстрактный указатель на интерфейс, используйте *IUnknown**. Самый гибкий метод — передача клиентом IID, и именно так работает *QueryInterface*:

```
HRESULT GetIFace([in] const IID& iid,[out, iid_is(iid)] IUnknown** ppi);
```

Здесь атрибут *iid_is* используется для указания MIDL идентификатора интерфейса. Конечно, вместо этого можно было бы использовать:

```
HRESULT GetMyInterface([out] IMyInterface** pIMy);
```

Но что произойдет, если будет возвращен *IMy2* или *IMyNewAndVastlyImproved*?

Компилятор MIDL

Теперь, когда у нас есть файл IDL, его можно пропустить через компилятор MIDL, который сгенерирует несколько файлов. Если описания наших интерфейсов находятся в файле FOO.IDL, то скомпилировать этот файл можно следующей командой:

```
midl foo.idl
```

В результате будут сгенерированы файлы, перечисленные в табл. 10-1.

Таблица 11.1 *Файлы, гененрируемые компилятором MIDL*

Имя файла	Содержимое
FOO.H	Заголовочный файл (для C и C++), содержащий объявления всех интерфейсов, описанных в файле IDL. Имя заголовочного файла можно изменить с помощью параметра командной строки */header* или */h*.
FOO_I.C	Файл C, в котором определены все GUID, использованные в файле IDL. Имя файла можно изменить с помощью параметра командной строки */iid*.
FOO_P.C	Файл C, реализующий код заместителей и заглушек для всех описанных в файле IDL интерфейсов. Имя файла можно изменять с помощью параметра командной строки */proxy*.
DLLDATA.C	C Файл C, реализующий DLL, которая содержит код заместителей и заглушек. Имя файла можно изменить с помощью параметра командной строки */dlldata*.

Если в файле IDL имеется ключевое слово *library*, то по приведенной выше команде будет сгенерирована библиотека типа. (Как Вы помните, более подробно библиотеки типа будут рассматриваться в восхитительной следующей главе этой книги.) На рис. 11.4

показаны файлы, генерируемые компилятором MIDL. Здесь также показано, как из этих файлов генерируется DLL заместителя, — процесс, который мы рассмотрим чуть ниже.

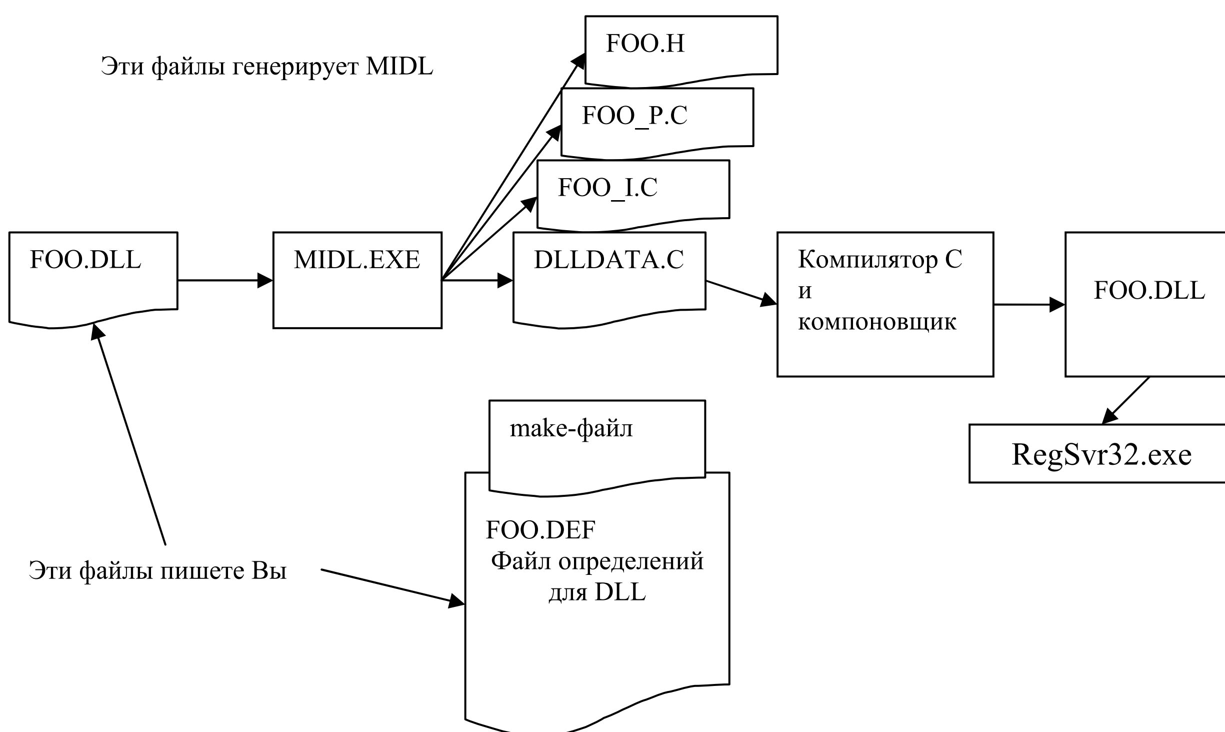


Рис. 11.4 *Получение и использование файлов, генерируемых компилятором MIDL*

Сборка примера программы

Чтобы наш разговор был более предметным, давайте соберем пример программы для этой главы. Все необходимые файлы есть на прилагающемся к книге компакт-диске. С помощью make-файла примера можно построить две версии сервера компонента: SERVER.DLL и SERVER.EXE. Для того, чтобы построить обе версии сразу, используется команда

```
nmake –f makefile
```

MAKEFILE дважды вызывает файл MAKE-ONE для сборки двух разных версий сервера. Промежуточные файлы сервера внутри процесса будут помещены в подкаталог \INPROC. Промежуточные файлы сервера вне процесса будут помещены в подкаталог \OUTPROC. В make-файлах этого примера для запуска MIDL используется следующая командная строка:

```
midl /h iface.h /iid guids.c /proxy proxy.c server.idl
```

Эта команда переименовывает файлы, генерируемые MIDL, чтобы мы могли использовать прежние имена. Вместо того, чтобы писать определения интерфейса и в IFACE.H, и в SERVER.IDL, мы создаем только SERVER.IDL, а компилятор MIDL по нему генерирует IFACE.H автоматически. Точно так же нам более нужны GUID в файле GUID.CPP. Теперь мы просто подключаем GUIDS.C.

Заголовочный файл, генерируемый MIDL, можно использовать в программах как на C, так и на C++. Единственный недостаток этих заголовочных файлов в том, что они практически нечитабельны. Вы поймете, что имеется в виду, если посмотрите на содержимое сгенерированного MIDL файла IFACE.H. Как видите, его нелегко

расшифровать. Тем не менее, это гораздо лучше, чем вручную поддерживать одинаковые описания интерфейса в разных местах.

Сборка DLL заместителя

Чтобы получить DLL заместителя/заглушки, нужно откомпилировать и скомпоновать файлы C, сгенерированные MIDL. Компилятор MIDL генерирует для нас код на C, который реализует для наших интерфейсов заместители и заглушки. Однако мы по-прежнему должны сами скомпилировать эти файлы в DLL. Первый шаг — написать для DLL заглушку файла DEF. Это очень просто. Файл DEF, который я использую, приведен ниже.

```
LIBRARY Proxy.dll
DESCRIPTION 'Proxy/Stub DLL'
EXPORTS
      DllGetClassObject @1 PRIVATE
      DllCanUnloadNow @2 PRIVATE
      GetProxyDllInfo @3 PRIVATE
      DllRegisterServer @4 PRIVATE
      DllUnregisterServer @5 PRIVATE
```

Теперь осталось все это откомпилировать и скомпоновать. Как это сделать, показывает следующий фрагмент файла MAKE-ONE:

```
iface.h server.tlb proxy.c guids.c dlldata.c : server.idl
midl /h iface.h /iid guids.c /proxy proxy.c server.idl

!IF "$(OUTPROC)" != ""
dlldata.obj : dlldata.c
      cl /c /DWIN32 /DREGISTER_PROXY_DLL dlldata.c

proxy.obj : proxy.c
      cl /c /DWIN32 /DREGISTER_PROXY_DLL proxy.c

PROXYSTUBOBJS = dlldata.obj \
                proxy.obj \
                guids.obj

PROXYSTUBLIBS = kernel.lib \
                rpcndr.lib \
                rpcns4.lib \
                rpcrt4.lib \
                uuid.lib

proxy.dll : $(PROXYSTUBOBJS) proxy.def
  link /dll /out:proxy.dll /def:proxy.def \
    $(PROXYSTUBOBJS) $(PROXYSTUBLIBS)

regsvr32 /s proxy.dll
```

Регистрация DLL заместителя/заглушки

Обратите внимание, что код make-файла определяет символ REGISTER_PROXY_DLL при компиляции файлов DLLDATA.C и PROXY.C. В результате генерируется код, позволяющий DLL заместителя/заглушки выполнять саморегистрацию. Затем, после компоновки DLL заместителя, make-файл регистрирует ее. Тем самым гарантируется, что Вы не забудете зарегистрировать DLL заместителя. Если

бы Вы забыли это сделать, то несколько часов удивлялись бы, отчего вдруг не работает программа. Я это испытал.

Что именно DLL заместителя/заглушки помещает в Реестр? Давайте рассмотрим наш пример. Убедитесь, что Вы скомпоновали программу; код make-файла автоматически регистрирует заместитель и сервер, так что Вам делать это нет необходимости. Или же запустите файл REGISTER.BAT для регистрации скомпилированной ранее версии программы.

Теперь давайте запустим старый верный REGEDIT.EXE и посмотрим на раздел Реестра:

```
HKEY_CLASSES_ROOT\
                  Interface\
                        {32BB8323-B41B-11CF-A6BB-0080C7B2D682}
```

Приведенный выше GUID — это IID интерфейса *IX*. В этом разделе содержится несколько записей. Самая для нас интересная — *ProxyStubClsid32*. В этом разделе содержится CLSID DLL заместителя/заглушки интерфейса; для интерфейсов *IX*, *IY* и *IZ* он совпадает. Если найдете этот CLSID в разделе HKEY_CLASSES_ROOT\CLSID, там можно обнаружить и подраздел *InprocServer32*, который указывает на PROXY.DLL. Как видите, интерфейсы регистрируются независимо от реализующих их компонентов (рис. 11.5).

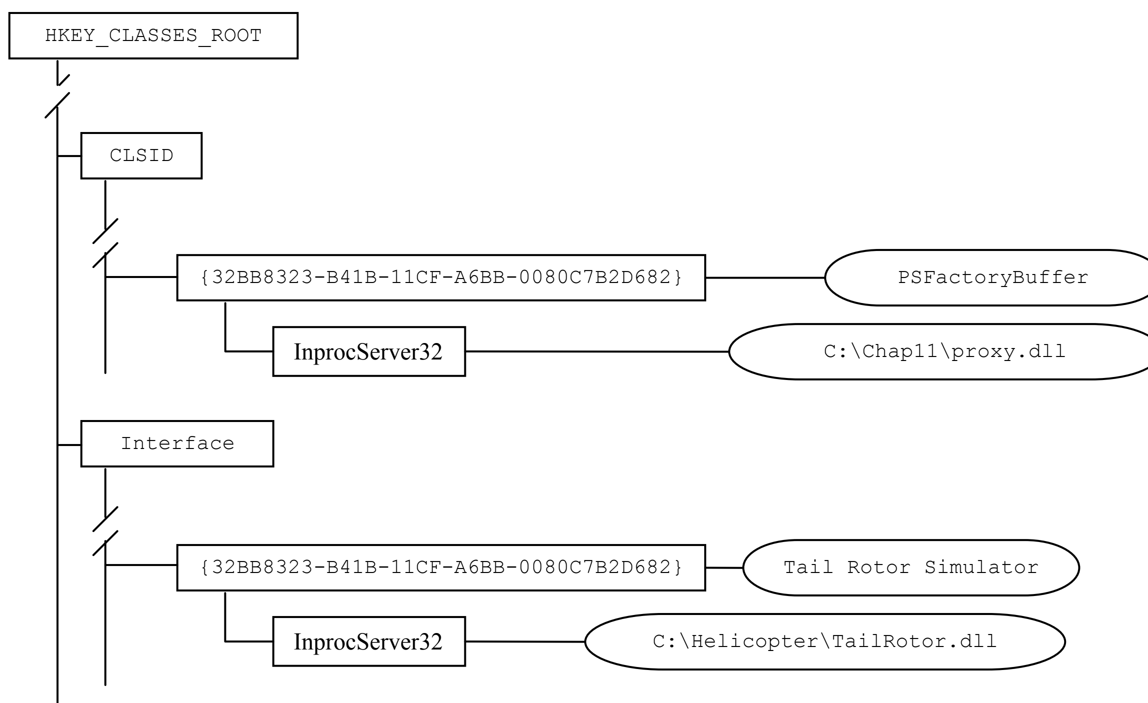


Рис. 11.5 *Структура информации, добавляемой в Реестр кодом заместителя/заглушки, сгенерированным MIDL*

При помощи MIDL мы можем вызывать функции и выполнять маршалинг параметров через границы процессов — и все будет выглядеть так же, как и при вызове компонента внутри процесса.

Реализация локального сервера

Теперь пришло время рассмотреть изменения в *CFactory*, необходимые для поддержки серверов вне процесса. Всякий раз, пересекая границу, Вы должны быть

готовы изменить свои привычки и поведение, чтобы соответствовать местным обычаям. Точно так же обслуживание компонента из EXE отличается от обслуживания компонента из DLL. Поэтому мы должны изменить *CFactory*, чтобы она обслуживала как компоненты в DLL, так и компоненты в EXE. Мы также внесем небольшие изменения в *CUnknown*. Однако код самих компонентов останется тем же самым.

В коде используется символ _OUTPROC_SERVER_, помечающий фрагменты, специфичные для локальных серверов (когда символ определен) или для серверов внутри процесса (когда он не определен). Прежде чем перейти к рассмотрению изменений в *CFactory*, давайте запустим пример программы.

Работа примера программы

При запуске клиент запросит Вас, хотите ли Вы использовать версию компонента для сервера внутри или вне процесса. Для подключения к компоненту внутри процесса клиент использует CLSCTX_INPROC_SERVER, а для подключения к компоненту вне процесса — CLSCTX_LOCAL_SERVER.

Если Вы решите использовать компонент, реализованный сервером внутри процесса, то все будет работать в точности, как в предыдущей главе. Однако если Вы выберете сервер вне процесса, программа будет работать несколько иначе. Первое, что Вы заметите, — вывод на экран теперь идет только от клиента. Это связано с тем, что компонент в другом процессе использует не то консольное окно, что клиент.

Вместо того, чтобы просто запустить клиент, сначала запустим сервер из командной строки. Дважды щелкните значок SERVER.EXE или воспользуйтесь командой *start*:

```
C:\>start server
```

Сервер начнет выполняться, и на экране появится его окно. Теперь запустите клиент и прикажите ему подключиться к локальному серверу. Клиент будет посылать сой вывод в новое консольное окно, а вывод локального сервера пойдет в его собственное окно.

Нет точек входа

Давайте теперь демистифицируем поведение этого примера. EXE не могут экспортировать функции. Наши серверы внутри процесса зависелт от наличия следующих экспортированных функций:

```
DllCanUnloadNow
DllRegisterServer
DllUnregisterServer
DllGetClassObject
```

Теперь нам нужна замена для этих функций. Заменить *DllCanUnloadNow* легко. EXE, в отличие от DLL, не является пассивным модулем — он управляет своей жизнью сам. EXE может отслеживать счетчик блокировок и, когда тот станет равным 0, выгрузить себя. Следовательно, для EXE нет необходимости реализовывать *DllCanUnloadNow*. Вычеркиваем ее из списка.

Следующие две функции — *DllRegisterServer* и *DllUnregisterServer* — заменить почти так же просто. EXE поддерживают саморегистрацию путем обработки параметров командной строки *RegServer* и *UnRegServer*. Все, что должен сделать наш локальный сервер, — это при получении соответствующего параметра командной строкивызвать *CFactory::RegisterAll* или *CFactory::UnregisterAll*. Пример кода, выполняющего эти действия, можно

найти в файле OUTPROC.CPP. (Попутно замечу, что локальный сервер регистрирует местоположение своегоEXE в разделе *LocalServer32*, а не в разделе *InprocServer32*. Вы можете заметить соответствующее изменение в файле REGISTRY.CPP.)

Таким образом, у нас осталась только *DllClassObject*, заменить которую несколько труднее, чем остальныефункции, экспортируемые DLL.

Запуск фабрик класса

Возвращаясь к гл. 8, вспомните, что *CoCreateInstance* вызывает *CoGetClassObject*, которая вызывает *DllGelClassObject*. Последняя возвращает указатель на *IClassFactory*, который используется для создания компонента. Поскольку EXE не могут экспортировать *DllGetClassObject*, нужен другой способ передачи *CoGetClassObject* нашего указателя на *IClassFactory.*

Решение, предлагаемое COM, — поддержка внутренней таблицы зарегистрированных фабрик класса. Когда клиент вызывает *CoGetClassObject* с соответствующими параметрами, COM сначала просматривает свою внутреннюю таблицу фабрик класса, ища заданный клиентом CLSID. Если фабрика класса в таблице отсутствует, то COM обращается к Реестру и запускает соответствующий модуль EXE. Задача последнего — как можно скорее зарегистрировать свои фабрики класса, чтобы их могла найти COM. Для регистрации фабрики класса EXE использует функцию COM *CoRegisterClassObject*. При запуске EXE обязан зарегистрировать все поддерживаемые им фабрики. Я добавил в *CFactory* новую стратегическую функцию-член *StartFactories*, которая вызывает *CoRegisterClassObject* для каждого компонента в массиве структур *CFactoryData*. Код этой функции приведен ниже.

```
BOOL CFactory::StartFactories()
{
  CFactoryData* pStart = &g_FactoryDataArray[0];
  const CFactoryData* pEnd =
  &g_FactoryDataArray[g_cFactoryDataEntries - 1];
  for(CFactoryData* pData = pStart; pData <= pEnd; pData++)
  {
    // Инициализировать указатель и признак фабрики класса
    pData->m_pIClassFactory = NULL;
    pData->m_dwRegister = NULL;

    // Создать фабрику класса для компонента
    IClassFactory* pIFactory = new CFactory(pData);

    // Зарегистрировать фабрику класса
    DWORD dwRegister;
    HRESULT hr = ::CoRegisterClassObject(
                     *pData->m_pCLSID,
                     static_cast<IUnknown*>(pIFactory),
                     CLSCTX_LOCAL_SERVER,
                     REGCLS_MULTIPLEUSE,
                     &dwRegister);
    if (FAILED(hr))
    {
      pIFactory->Release();
      return FALSE;
    }
    // Запомнить информацию
    pData->m_pIClassFactory = pIFactory;
    pData->m_dwRegister = dwRegister;
  }
  return TRUE;
}
```

Данный код использует две новых переменных-члена, которые я добавил в класс *CfactoryData*. Переменная *m_pIClassFactory* содержит указатель на работающую фабрику класса для CLSID, хранящегося в *m_pCLSID*. Переменная *m_dwRegister* содержит магический признак (cookie)1 для данной фабрики.

Как видите, для регистрации фабрики класса нужно лишь ее создать и передать указатель на ее интерфейс функции *CoRegisterClassObject*. Значение большинства параметров *CoRegisterClassObject* легко понять из приведенного выше кода. Сначала идет ссылка на CLSID регистрируемого класса, за которой следует указатель на фабрику класса. Магический признак возвращается через последний параметр; он используется для отзыва фабрики класса с помощью функции *CoRevokeClassObject*. Третий и четвертый параметр — это флажки, управляющие поведением *CoRegisterClassObject.*

Флажки для *CoRegisterClassObject*

Третий и четвертый параметр этой функции используются вместе, и смысл одного изменяется в зависимости от значения другого. В результате интерпретация становиться весьма запутанной.

Четвертый параметр указывает, может ли один экземпляр данного EXE обслуживать более одного экземпляра соответствующего компонента. Проще всего это понять, сравнив сервер EXE с приложением SDI (single document interface — однодокументный интерфейс). Для загрузки нескольких документов необходимо запустить несколько экземпляров такого приложения, тогда как один экземпляр приложения MDI (multiple document interface — многодокументный интерфейс) может открыть несколько документов. Если Ваш сервер EXE похож на приложение SDI, в том смысле, что он может обслуживать только один компонент, следует задать REGCLS_SINGLEUSE и CLSCTX_LOCAL_SERVER.

Если сервер EXE может поддерживать несколько экземпляров компонента, подобно тому, как приложение MDI может открыть несколько документов, используйте REGCLS_MULTI_SEPARATE:

```
hr = ::CoRegisterClassObject(clsid, pUnknown,
            CLSCTX_LOCAL_SERVER,
            REGCLS_MULTI_SEPARATE,
            &dwRegister);
```

Возникает интересная ситуация. Предположим, что наш EXE-модуль зарегистрировал несколько компонентов. Пусть, кроме того, этому EXE необходимо использовать один из зарегистрированных им компонентов. Если соответствующая фабрика класса зарегистрирована с помощью приведенного выше оператора, то для обслуживания компонента будет запущен еще один экземпляр EXE. Очевидно, что в большинстве случаев это не столь эффективно, как мы бы хотели. Для регистрации сервера EXE как сервера своих собственных компонентов внутри процесса, объедините, как показано ниже, флаг CLSCTX_LOCAL_SERVER с флагом CLSCTX_INPROG_SERVER:

```
hr = ::CoRegisterClassObject(clsid, pUnknow,
            CLSCTX_LOCAL_SERVER | CLSCTX_INPROC_SERVER,
            REGCLS_MULTI_SEPARATE,
            &dwRegister);
```

В результате объединения флажков сервер EXE сможет самостоятельно обслуживать свои компоненты. Поскольку данный случай наиболее распространен, для автоматического включения CLSCTX_INPROC_SERVER при заданном

CLSCTX_LOCAL_SERVER используется специальный флаг REGCLS_MULTIPLEUSE. Ниже приведен эквивалент предыдущего вызова:

```
hr = ::CoRegisterClassObject(clsid, pUnknown,
            CLS_LOCAL_SERVER,
            REGCLS_MULTIPLEUSE,
            &dwRegister);
```

изменив пример программы, можно увидеть различие между REGCLS_MULTIPLEUSE и REGCLS_MULTI_SEPARATE. Сначала удалите информацию сервера внутри процесса из Реестра следующей командой:

```
regsvr32 /u server.dll
```

Это гарантирует, что единственным доступным сервером будет локальный. Затем запустите клиент и выберите второй вариант для активации локального сервера. Локальный сервер будет прекрасно работать. Обратите внимание, что в функциях *Unit* в файлах CMPNT1.CPP и CMPNT2.CPP мы создаем компонент, используя CLSCTX_INPROC_SERVER, — но ведь мы только что удалили информацию сервера внутри процесса из Реестра! Следовательно, наш EXE сам предоставляет себе внутрипроцессные версии этих компонентов.

Теперь заменим REGCLS_MULTIPLEUSE на REGCLS_MULTU_SEPARATE и *CFactory::StartFactories*. (Строки, которые нужно изменить, помечены в CFACTORY.CPP символами *@Multi*.) Скомпонуйте клиент и сервер заново, запустите клиент и выберите второй вариант. Вызов создания компонента потерпит неудачу, так как создания внутренних компонентов нет сервера внутри процесса, а REGCLS_MULTI_SEPARATE заставляет COM отвергать попытки сервера самостоятельно обслуживать компоненты внутри процесса.

Остановка фабрик класса

Когда работа сервера завершается, фабрики класса следует удалить из внутренней таблицы COM. Это выполняется при помощи функции библиотеки COM *CoRevokeClassObject*. Метод *StopFactories* класса *Cfactory* вызывает *CoRevokeClassObject* для всех поддерживаемых данных EXE фабрик класса:

```
void CFactory::StopFactories()
{
  CFactoryData* pStart = &g_FactoryDataArray[0];
  const CFactoryData* pEnd =
  &g_FactoryDataArray[g_cFactoryDataEntries - 1];
  for (CFactoryData* pData = pStart; pData <= pEnd; pData++)
  {
    // Прекратить работу фабрики класса с помощью магического признака.
    DWORD dwRegister = pData->m_dwRegister;
    if (dwRegister != 0)
    {
      ::CoRevokeClassObject(dwRegister);
    }
    //Освободить фабрику класса.
    IClassFactory* pIFactory = pData->m_pIClassFactory;
    if (pIfactory != NULL)
    {
      pIFactory->Release();
    }
  }
}
```

Обратите внимание, что *CoRevokeClassObject* передается пресловутый магический признак, который мы получили ранее от *CoRegisterClassObject.*

Изменения в LockServer

Серверы внутри процесса экспортируют функцию *DllCanUnloadNow*. Библиотека COM вызывает ее, чтобы определить, можно ли выгрузить сервер из памяти. *DllCanUnloadNow* реализована при помощи статической функции *CFactory::CanUnloadNow*, которая проверяет значение статической переменной *CUnknown::s_ActiveComponents*. Всякий раз при создании нового компонента этот счетчик увеличивается. Однако, как обсуждалось в гл. 8, мы неувеличиваем его значение при создании новой фабрики класса. Следовательно, сервер допускает завершение своей работы даже при наличии у него активных фабрик класса.

Теперь должно быть понятно, почему мы не учитывали фабрики класса вместе с активными компонентами. Первое, что делает локальный сервер, это создает свои фабрики класса; последнее, что он делает, — удаляет их.Если бы для завершения работы серверу нужно было дожидаться ликвидации этих фабрик, ждать ему пришлось бы долго — потому что именно он и должен их ликвидировать перед окончанием работы. Поэтому клиент должен использовать функцию *IClassFactory::LockServer*, если он хочет гарантировать, что сервер присутствует в памяти, пока клиент пытается создавать компоненты.

Нам необходимо внести некоторые изменения в *LockServer*, чтобы использовать эту функцию в локальном сервере. Позвольте мне пояснить необходимость изменений. DLL не управляет временем своей жизни. EXE загружает DLL, и EXE выгружает DLL. Однако EXE управляют временем своего существования и *могут* выгружаться сами. Никто не будет выгружать модуль EXE, он должен делать это сам. Следовательно, нам необходимо изменить *LockServer*, чтобы завершить работу EXE, когда счетчик блокировок становиться равным нулю. Я добавил к *CFactory* новую функцию-член *CloseExe*, которая посылает WM_QUIT в цикл выборки сообщений приложения:

```
#ifdef _OUTPROC_SERVER_
void CFactory::CloseExe()
{
  if (CanUnloadNow() == S_OK)
  {
    ::PostThreadMessage(s_dwThreadID, WM_QUIT, 0,0);
  }
}
#else
// CloseExe ничего не делает для сервера внутри процесса.
void CFactory::CloseExe() { /*Пусто*/ }
#endif
```

Заметьте, что для сервера внутри процесса эта функция ничего не делает. Чтобы сделать код изумительно эффективным, я просто вызываю *CloseExe* из *LockServer.*

```
HRESULT __stdcall CFactory::LockServer(BOOL block)
{
  if (block)
  {
    ::InterlockedIncrement(&s_cServerLocks);
  }
  else
  {
    ::InterlockedDecrement(&s_cServerLocks);
  }
```

```
  // Для сервера вне процесса проверить, можно ли завершить работу программы.
  CloseExe();
  return S_OK;
}
```

необходимо также вызывать *CloseExe* из деструкторов компонентов; это еще одно место, где модуль EXE может определить, нужно ли ему завершить работу. Для этого я изменил деструктор *Cunknown:*

```
CUnknown::~CUnknown()
{
  ::InterlockedDecrement(&s_cActiveComponents);
  //Если это сервер EXE, завершить работу.
  CFactory::CloseExe();
}
```

Цикл сообщений цикл сообщений цикл сообщений…

В программах на C и C++ есть стандартная точка входа, которая называется *main*. С функции *main* начинается выполнение программы. Программа завершает работу, когда происходит возврат из *main*. Точно так же в программах для Windows есть функция *WinMain*. Таким образом, чтобы модуль EXE не прекращал работу, необходим цикл, предотвращающий выход из *main* или *WinMain*. Так как наш сервер компонента работает под Windows, я добавил цикл выборки сообщений Windows. Он представляет собой упрощенную версию цикла, используемого всеми программами для Windows.
Код цикла выборки сообщений содержится в файле OUTPROC.CPP. Компиляция данного файла и компоновка с ним происходят только в том случае, если собирается версия сервера вне процесса.

Подсчет пользователей

Помните, как мы запускали сервер перед запуском клиента? После завершения работы клиента сервер оставался загруженным. Пользователи сервера — также клиенты, и у них должен быть свой счетчик блокировок. Поэтому, когда пользователь создает компонент, мы увеличиваем *CFactory::s_cServerLocks*. Таким образом, сервер будет оставаться в памяти, пока с ним работает пользователь.

Как нам определить, что сервер запустил пользователь, а не библиотека COM? Когда *CoGetClassObject* загружает EXE локального сервера, она задает в командной строке аргумент *Embedding*. EXE проверяет наличие этого аргумента в командной строке. Если *Embedding* там нет, то сервер увеличивает *s_cServerLocks* и отображает окно для пользователя. Когда пользователь завершает работу сервера, с тем по-прежнему могут работать клиенты. Следовательно, когда пользователь завершает программу, сервер должен убрать с экрана пользовательский интерфейс, но не завершаться, пока не закончит обслуживание всех клиентов. Таким образом, сервер не должен посылать себе сообщение WM_QUIT при обработке сообщения WM_DESTROY, если только *CanUnloadNow* не возвращает S_OK.

Удаленный сервер

Самое замечательное в локальном сервере, который мы реализовали в этой главе, — то, что он является и удаленным сервером. Без каких-либо изменений CLIENT.EXE и SERVER.EXE могут работать друг с другом по сети. Для этого Вам потребуется по крайней мере два компьютера, на которых работает Windows NT 4.0 или Windows 95 с

установленной поддержкой DCOM. Естественно, эти компьютеры должны быть соединены между собой сетью.

Чтобы заставить клиента использовать удаленный сервер, воспользуемся программой конфигурации DCOMDCOMCNFG.EXE, которая входит в состав Windows NT. Эта программа позволяет изменять различные параметры приложений, установленных на компьютере, в том числе и то, исполняются ли они локально или удаленно.

В табл. 11.2 представлены пошаговые инструкции для выполнения SERVER.EXE в удаленном режиме.

Таблица 11.2 *Запуск SERVER.EXE на удаленной машине*

Действие	Локальный компьютер	Удаленный компьютер
Скомпонуйте CLIENT, SERVER.EXE и PROXY.DLL с помощью команды *nmake-f makefile*. Если Вы уже их скомпоновали, делать это заново не нужно.	*v*	
Скопируйте CLIENT.EXE, SERVER.EXE и PROXY.DLL на удаленный компьютер.		*v*
Зарегистрируйте локальный сервер с помощью команды *server /RegServer*.	*v*	*v*
Зарегистрируйте заместитель с помощью команды *regsvr32 Proxy.dll*.	*v*	*v*
Запустите CLIENT.EXE и выберите вариант локального сервера. Это позволит Вам убедиться, что программы работают на обоих компьютерах.	*v*	*v*
Запустите DCOMCNFG.EXE. Выберите компонент Inside COM Chapter 10 Example Component 1 и щелкните Properties. Выберите вкладкуLocation. Отключите опцию *Run Application On This Computer* и выберите опцию *Run Application On Following Computer*. Введите имя удаленного компьютера, на котором будет выполняться SERVER.EXE. Щелкните вкладку *Identity* и выберите кнопку-переключатель Interactive User.	*v*	
В зависимости от Ваших прав доступа может потребоваться изменить установки на вкладке Security.	*v*	*v*
Запустите SERVER.EXE, чтобы увидеть его вывод на экран.		*v*
Запустите CLIENT.EXE и выберите вариант 2, чтобы использовать локальный сервер компонента.	*v*	
В окне SERVER.EXE должны появиться сообщения.		*v*
Сообщения также должны появиться в консольном окне CLIENT.EXE.	*v*	

Я нахожу поистине восхитительным, что с помощью служебной программы мы можем превратить локальный сервер в удаленный. Остается вопрос — как это работает?

Что делает DCOMCNFG.EXE?

Если после запуска DCOMCNFG.EXE Вы запустите на той же машине REGEDIT.EXE, то сможите увидеть часть этого волшебства в Реестре. Найдите следующий раздел Реестра:

```
HKEY_CLASSES_ROOT\
                  CLSID\
                        {0C092C29-882C-11CA-A6BB-0080C7B2D682}
```

В дополнение к дружественному имени компонента Вы увидите новое значение с именем *AppID.* CLSID идентифицирует компонент, и соответствующий раздел Реестра содержит информацию о нем. В разделе *LocalServer32* указан путь к приложению, в котором реализован компонент, но CLSID никак больше не связан с приложением. Однако DCOM нужно связать с приложением, содержащим компонент, определенную информацию. Для этого используется *AppID*.

Значением *AppID*, также как и CLSID, является GUID. Информация об *AppID* хранится в ветви Реестра *AppID*; и здесь аналогично CLSID. Информацию об *AppID* для SERVER.EXE можно найти в разделе Реестра:

```
HKEY_CLASSES_ROOT\
                  AppID\
                        {0C092C29-882C-11CA-A6BB-0080C7B2D682}
```

В разделе для *AppID* хранятся как минимум три значения. Значение по умолчанию — дружественное имя. Другие именованные значения — *RemoteServerName*, задающее имя сервера, на котором находится приложение, и *RunAs*, сообщающее DCOM, как исполнять приложение. Соответствующая структура Реестра показана на рис. 11.6.

Кроме этого, непосредственно в разделе *AppID* хранится имя приложения. Вы должны увидеть в Редакторе Реестра такой раздел:

```
HKEY_CLASSES_ROOT\
                  AppID\
                        server.exe
```

В нем только одно именованное значение, которое указывает обратно на *AppID*.

Но как это работает?

Внесение записей в Реестр дает мало пользы до тех пор, пока у нас нет кода, который их читает. DCOM расширяет библиотеку COM, включая в нее свою реализацию функции *CoGetClassObject*. Эта функция не только гораздо мощнее, но и гораздо запутанней. *CoGetClassObject* может работать множеством разных способов.

Обычно она принимает CLSID и открывает сервер компонента в соответствующем контексте. Если контекстом является CLSCTX_REMOTE_SERVER, *CoGetClassObject* отыскивает компонент в Реестре и проверяет, задан ли для него *AppID*. В этом случае функция отыскивает в Реестре значение *RemoteServerName*. Если имя сервера найдено, то *CoGetClassObject* пытается запустить сервер удаленно. Именно это и происходило в примере выше.

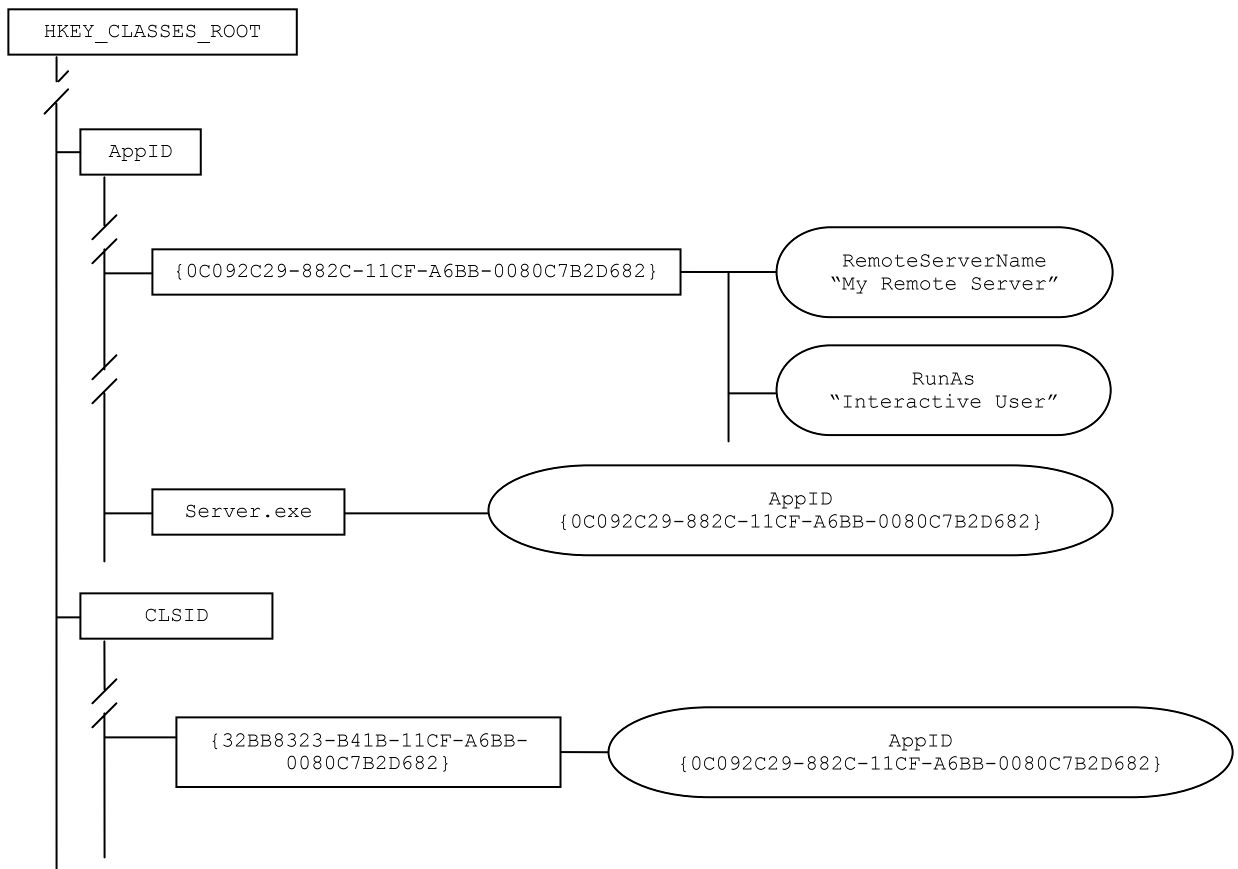


Рис. 11.6 *Организация записей Реестра для AppID.*

Другая информация DCOM

Хотя, перемещаясь по Реестру, можно превратить локальный сервер в удаленный, можно также программно указать, что Вам нужен доступ к удаленному серверу. Для этого следует заменить *CoCreateInstance* на *CoCreateInstanceEx* или модифицировать вызов *CoGetObject*. Ниже приведен пример использования *CoCreateInstanceEx* для создания удаленного компонента:

```
// Создать структуру для хранения информации о сервере.
COSERVERINFO ServerInfo;

// Инициализировать структуру нулями.
memset(&ServerInfo, 0, sizeof(ServerInfo));

// Задать имя удаленного сервера.
ServerInfo.pwszName = L”MyRemoveServer”;

// Подготовить структуры MULTI_QI для нужных нам интерфейсов.
MULTI_QI mqi[3];
mqi[0].pIID = IIDX_IX; // [in] IID требуемого интерфейса
mqi[0].pItf = NULL;    // [out] Указатель интерфейса
mqi[0].hr = S_OK;      // [out] Результат вызова QI для интерфейса
mqi[1].pIID = IIDX_IY;
mqi[1].pItf = NULL;
mqi[1].hr = S_OK;
mqi[2].pIID = IIDX_IZ;
mqi[2].pItf = NULL;
mqi[2].hr = S_OK;
```

```
HRESULT hr = CoCreateInstanceEx(CLSID_Component1,
                        NULL,
                        CLSCTX_REMOTE_SERVER,
                        &ServerInfo,
                        3, // Число интерфейсов
                        &mqi);
```

Первое бросающееся в глаза отличие *CoCreateInstanceEx* от *CoCreateInstance* — то, что первая функция принимаетв качестве параметра структуру COMSERVERINFO, содержащую имя удаленного сервера. Однако самый интересный аспект *CoCreateInstanceEx* — структура MULTI_QI.

MUTI_QI

Для компонентов внутри процесса вызовы *QueryInterface* выполняются очень быстро. *QueryInterface* достаточно быстро работает и для локальных серверов. Но когда необходимо пересылать информацию по сети, накладные расходы на любой вызов функции значительно возрастают. Работа приложения может и вовсе застопориться в результате повторяющихся вызовов, включая вызовы *QueryInterface*. В связи с этим для сокращения накладных расходов на вызовы *QueryInterface* в DCOM определена новая структура с именем MULTI_QI. Эта структура позволяет выполнять запрос нескольких интерфейсов за один раз, что может существенно уменьшить накладные расходы.

В приведенном выше пример мы запрашиваем интерфейсы *IX,IY* и *IZ* одновременно. *CoCreateInstanceEx* возвращает S_OK, если ей удалось получить все интерфейсы, заданные структурами MULTI_QI. Она возвращает E_NOINTERFACE, если не удалось получить ни одно интерфейса. Если же получены некоторые, но не все требуемые интерфейсы, возвращается CO_S_NOTALLINTERFACES.

Код ошибки, связанный с каждым отдельным интерфейсом, записывается в поле *hr* структуры MULTI_QI. Указатель на интерфейс возвращается в поле *pItf*.

Чтобы запросить несколько интерфейсов, *CoCreateInstanceEx* запрашивает у компонента после его создания интерфейс *ImultiQI*. Он объявлен так:

```
interface IMultiQI : IUnknown
{
  virtual HRESULT __stdcall QueryMultipleInterfaces
  (ULONG interfaces,
  MULTI_QI* pMQUIs);
};
```

Самое замечательное то, что Вам не нужно реализовывать *IMultiQI* для своего компонента. Удаленный заместитель компонента предоставляет этот интерфейс автоматически.

CoCreateInstance не работает под Windows 95

Если Вы определите символ препроцессора

```
_WIN32_DCOM
```

или

```
_WIN32_WINNT >= 0x0400,
```

то значения CSLCTX_ALL и CLSCTX_SERVER будут включать в себя CLSCTX_REMOTE_SERVER и не будут работать на системах Windows 95, где не

установлена поддержка DCOM. Если Вы разрабатываете программу для Windows 95 или Windows NT 3.51, убедитесь, что эти символы не определены.

Определение наличия DCOM

Для того, чтобы определить доступность сервисов DCOM, сначала проверьте, поддерживает ли OLE32.DLL свободные потоки. Если Ваша программа компонуется с OLE32.DLL статически, поступайте так:

```
if (GetProcAddress(GetModuleHandle("OLE32"), "CoInitializeEx") != NULL)
{
  // Свободные потоки поддерживаются.
}
```

Если Вы загружаете OLE32.DLL динамически, используйте следующий фрагмент кода:

```
hmodOLE32 = LoadLibrary("OLE32.DLL");
if (GetProcAddress(hmodOLE32, "CoInitializeEx") != NULL)
{
  // Свободные потоки поддерживаются.
}
```

Определив, что в системе имеется поддержка свободных потоков, проверьте, включена ли DCOM:

```
HKEY hKEY;
LONG lResult = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                        "SOFTWARE\\Microsoft\\Ole",
                        0,
                        KEY_ALL_ACCESS,
                        &hKey);

assert(lResult == ERROR_SUCCESS);
char rgch[2];
DWORD cb = sizeof(rgch);
LResult = RegQueryValueEx(hKey,
                        TEXT("EnableDCOM"),
                        0, NULL, rgch, &cb);

assert(lResult == ERROR_SUCCESS);
lResult = RegCloseKey(hKey);

assert(lResult == ERROR_SUCCESS);
if (rgch[0] == 'y' || rgch[0] == 'Y')
{
  // DCOM доступна
}
```

Резюме

Пересекать границы процессов — увлекательное занятие! Особенно когда у Вас много полезных инструментов (например, компилятор MIDL), которые облегчают эту задачу. Описав свои интерфейсы на IDL, мы можем с помощью MIDL сгенерировать необходимый код заглушки и заместителя для маршалинга интерфейсов через границы процесса. Еще более восхитительна встроенная в DCOM возможность превращать локальные серверы в удаленные простым изменением некоторых записей Реестра.

12. Диспетчерские интерфейсы и автоматизация

Как гласит пословица, «есть много способов содрать шкуру с кошки». Поскольку я никогда не пытался обдирать кошек и не нахожу в этом особого смысла, я предпочитаю говорить: «Есть много способов причесать кошку». Полагаю, что большинству кошек моя версия понравится больше. Один мой друг из Ла Гранде, штат Джорджия, использует другую версию этой фразы: «Есть много способов пахнуть как скунс». Если верить его матери, большинство этих способов ему известно. Все это говорит о том, как много можно придумать способов перефразировать поговорку.

В этой главе Вы увидите, что есть и много способов коммуникации между клиентом и компонентом. В предыдущих главах клиент использовал интерфейс COM для работы с компонентом напрямую. В этой главе мы рассмотрим Автоматизацию (в прошлом OLE Автоматизацию) — другой способ управления компонентом. Этот способ использует такие приложения, как Microsoft Word и Microsoft Excel, а также интерпретируемые языки типа Visual Basic и Java.

Автоматизация облегчает интерпретируемым языкам и макроязыкам доступ к компонентам COM, а также облегчает написание самих компонентов на этих языках. В Автоматизации делается упор на проверку типов во время выполнения за счет снижения скорости выполнения и проверки типов во время компиляции. Но если Автоматизация проста для программиста на макроязыке, то от разработчика на C++ она требует гораздо больше труда. Во многих случаях Автоматизация заменяет код, генерируемый компилятором, кодом, который написан разработчиком.

Автоматизация — не пристройка к COM, а надстройка над нею. *Сервер Автоматизации (Automation server)* — это компонент COM, который реализует интерфейс IDispatch. *Контролер Автоматизации (Automation Controller)*
— это клиент COM, взаимодействующий с сервером Автоматизации через интерфейс *IDispatch.* Контролер Автоматизации не вызывает функции сервера Автоматизации напрямую. Вместо этого он использует методы интерфейса *IDispatch* для неявного вызова функций сервера Автоматизации.

Интерфейс *IDispatch*, как и вся Автоматизация, разрабатывался для Visual Basic — чтобы его можно было использовать для автоматизации таких приложений, как Microsoft Word и Microsoft Excel. В конце концов из Visual Basic вырос Visual Basic for Applications — язык для Microsoft Office. Подмножество Visual Basic for Applications — Visual Basic Scripting Edition (VBScript) — можно использовать для автоматизации элементов управления на страницах Web. Версия 5.0 Microsoft Developer Studio использует VBScript в качестве своего макроязыка.

Практически любой сервис, который можно представить через интерфейсы COM, можно предоставить и при помощи *IDispatch*. Из этого следует, что *IDispatch* и Автоматизация — это не менее (а может быть, и более) широкая тема, чем COM. Поскольку эта книга посвящена все-таки COM, мы рассмотрим Автоматизацию только
частично. Но и этого все еще большая область: *IDispatch, disp*-интерфейсы, дуальные интерфейсы, библиотеки типа, IDL, VARIANT, BSTR и многое другое. По счастью, именно эти вопросы наиболее важны при программировании Автоматизации на C++.
Давайте начнем обдирать — то есть я хочу сказать причесывать — эту кошку, начиная с головы; посмотрим, чем работа через *IDispatch* отличается от работы через интерфейсы COM.

Новый способ общения

Что делает *IDispatch* столь замечательным интерфейсом COM? Дело в том, что *IDispatch* предоставляет клиентам и компонентам новый способ общения между собой. Вместо предоставления нескольких собственных интерфейсов, специфичных для его сервисов,

компонент может обеспечить доступ к этим сервисам через один стандартный интерфейс, *IDispatch.*

Прежде чем подробно рассматривать *IDispatch*, давайте разберемся, как он может поддерживать столь много функций; для этого мы сравним его со специализированными интерфейсами COM (которые он может заменить).

Старый способ общения

Давайте еще раз кратко рассмотрим прежний метод, используемый клиентами для управления компонентами. Возможно, Вас уже тошнит от этого, но клиент и компонент все же взаимодействуют через интерфейсы. Интерфейс представляет собой массив указателей на функции. Откуда клиенту известно, какой элемент массива содержит указатель на нужную функцию? Код клиента включает заголовочный файл, содержащий описание интерфейса как абстрактного базового класса. Компилятор считывает этот заголовочный файл и присваивает индекс каждому методу абстрактного базового класса. Этот индекс — индекс указателя на функцию в абстрактном массиве. Затем компилятор может рассматривать следующую строку кода:

```
pIX->FxStringOut(msg);
```

как

```
(*(pIX->pvtbl[IndexOfFxStringOut]))(pIX, msg);
```

где *pvtbl* — это указатель на ytbl данного класса, а *IndexOfFxStringOut* — индекс указателя на функцию *FxStringOut* в таблице указателей на функции. Все это происходит *автоматически* — Вы этого не знаете или, в большинстве случаев, Вас это не беспокоит.

Вам *придется* побеспокоиться об этом при разработке макроязыка для своего приложения. Макроязык будет гораздо мощнее, если сможет использовать компоненты COM. Но каким образом макроязык получит смещения функций в vtbl? Я сомневаюсь, что Вы захотите писать синтаксический анализатор C++ для разбора заголовочного файла интерфейса COM.

Когда макроязык вызывает функцию компонента COM, у него есть три элемента информации: ProgID компонента, реализующего функцию, имя функции и ее аргументы. Нам нужен простой способ, чтобы интерпретатор макроязыка мог вызывать функцию по ее имени. Именно для этого и служит *IDispatch.*

IDispatch, или «Я диспетчер, ты диспетчер…»

Говоря попросту, *IDipatch* принимает имя функции и выполняет ее. Описание *IDipatch* на IDL, взятое из файла OAIDL.IDL, приведено ниже:

```
interface IDispatch : IUnknown
{
  HRESULT GetTypeInfoCount([out] UINT * pctinfo);
  HRESULT GetTypeInfo([in] UINT iTInfo,
  [in] LCID lcid,
  [out] ItypeInfo ** ppTInfo);
  HRESULT GetIDsOfNames(
  [in] REFIID riid,
  [in, size_is(cNames)] LPOLESTR * rgszNames,
  [in] UINT cNames,
  [in] LCID lcid,
  [out, size_is(cNames)] DISPID * rgDispId);
  HRESULT Invoke([in] DISPID dispIdMember,
```

```
  [in] REFIID riid,
  [in] LCID lcid,
  [in] WORD wFlags,
  [in, out] DISPPARAMS * pDispParams,
  [out] VARIANT * pVarResult,
  [out] EXCEPINFO * pExcepInfo,
  [out] UINT * puArgErr);
};
```

Наиболее интересны в этом интерфейсе функции *GetIDsOfNames* и *Invoke*. Первая принимает имя функции и возвращает ее диспетчерский идентификатор, или DISPID. DISPID — это не GUID, а просто длинное целое (LONG), идентифицирующее функцию. DISPID не уникальны (за исключением данной реализации *IDipatch*). У каждой реализации *IDipatch* имеется собственный IID (некоторые называют его DIID).

Для вызова функции контроллер автоматизации передает ее DISPID функции-члену *Invoke*. Последняя использует DISPID как индекс в массиве указателей на функции, что очень похоже на обычные интерфейсы COM. Однако сервер Автоматизации не обязан реализовывать *Invoke* именно так. Простой сервер Автоматизации может использовать оператор *switch*, который выполняет разный код в зависимости от значения DISPID. Именно так реализовывали оконные процедуры, прежде чем стала популярна MFC.
У оконных процедур и *IDispatch::Invoke* есть другие общие черты. Как окно ассоциируется с оконной процедурой, так и сервер Автоматизации ассоциируется с функцией *IDispatch::Invoke*. Microsoft Windows посылает оконной процедуре сообщения; контроллер автоматизации посылает *IDispatch::Invoke* разные DISPID.

Поведение оконной процедуры определяется получаемыми сообщениями; поведение *Invoke* — получаемыми DISPID.

Способ действий *IDispatch::Invoke* напоминает и vtbl. *Invoke* реализует набор функций, доступ к которым осуществляется по индексу. Таблица vtbl — массив указателей на функции, обращение к которым также идет по индексу. Но если vtbl работает автоматически за счет магии компилятора C++, то *Invoke* работает благодаря
тяжкому труду программиста. Однако в C++ vtbl статические, и компилятор работает только во время компиляции. Если программисту C++ необходимо порождать vtbl во время выполнения, он предоставлен самому себе. С другой стороны, легко создать универсальную реализацию *Invoke*, которая сможет «на лету» адаптироваться для реализации самых разных сервисов.

Disp-интерфейсы

У реализации *IDispatch::Invoke* есть еще одно сходство с vtbl. Обе они определяют интерфейс. Набор функций, реализованных с помощью *IDispatch::Invoke*, называется *диспетчерским интерфейсом (dispatch interface)* или, короче, *disp-интерфейсом (dispinterface)*. По определению, интерфейс COM — это указатель на массив указателей на функции, первыми тремя из которых являются *QueryInterface, AddRef* и *Release*. В соответствии с более общим определением, интерфейс — это набор функций и переменных, посредством которых взаимодействуют две части программы. Реализация *IDispatch::Invoke* определяет набор функций, посредством которых взаимодействуют сервер и контроллер Автоматизации. Как нетрудно видеть, функции, реализованные
Invoke, образуют интерфейс, но не интерфейс COM.

На рис.12.1 диспетчерский интерфейс представлен графически. Слева изображен традиционный интерфейс COM — *IDispatch* реализованный при помощи vtbl. Справа показан disp-интерфейс. Центральную роль в disp-интерфейсе играют DISPID, распознаваемые *IDispatch::Invoke*. На рисунке показана одна из возможных реализаций *Invoke* и *GetIDsOfNames*: массив имен функций и массив указателей на функции, индексируемые DISPID. Это только один способ. Для больших disp-интерфейсов

GetIDsOfNames работает быстрее, если передаваемое ей имя используется в качестве ключа хеш-таблицы.

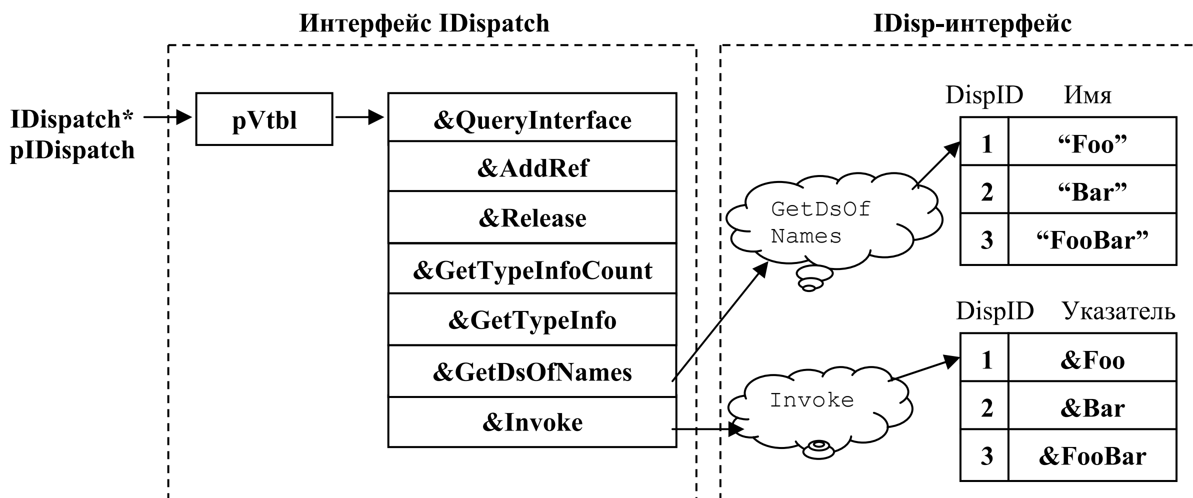


Рис. 12.1. *Disp-интерфейсы реализуются с помощью IDispatch и не являются интерфейсами COM. На этом рисунке представлена только одна из возможных реализаций IDispatch::Invoke*.

Конечно, для реализации *IDispatch::Invoke* можно использовать и интерфейс COM (рис. 12.2).

Дуальные интерфейсы

На рис. 12.2 представлен не единственный способ реализации disp-интерфейса при помощи интерфейса COM. Другой метод, показанный на рис. 12.3, состоит в том, чтобы интерфейс COM, реализующий *IDispatch::Invoke*, наследовал не *IUnknown*, а *IDispatch*. Так реализуют интерфейсы, называемые *дуальными интерфейсами (dual interface)*. Дуальный интерфейс — это disp-интерфейс, все члены которого, доступные через *Invoke*, доступны и напрямую через vtbl.

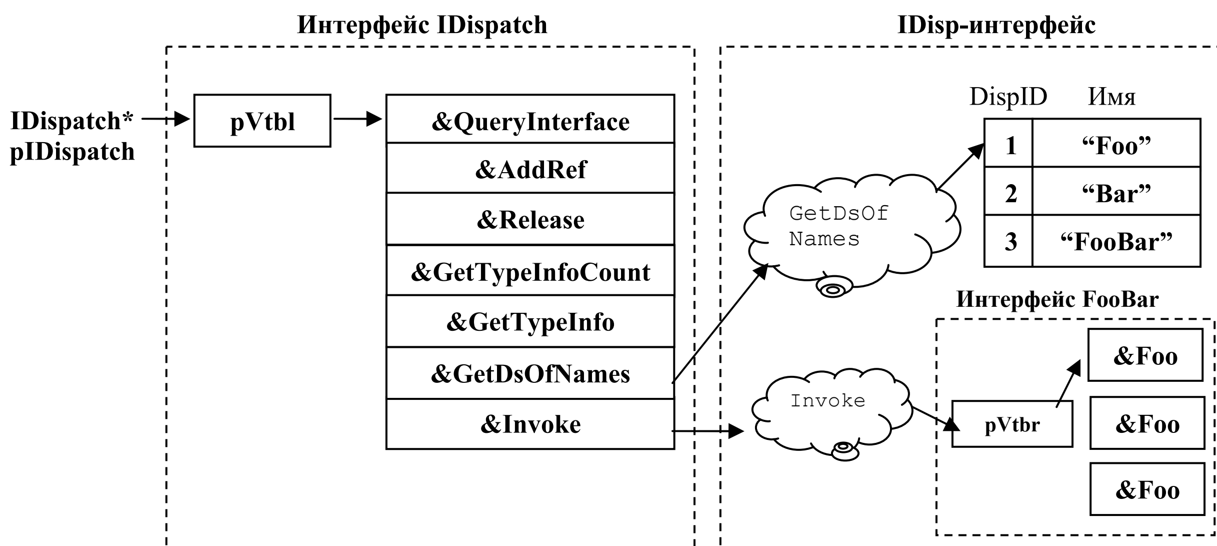


Рис. 11.2 *Реализация IDispatch::Invoke с помощью интерфейса COM.*

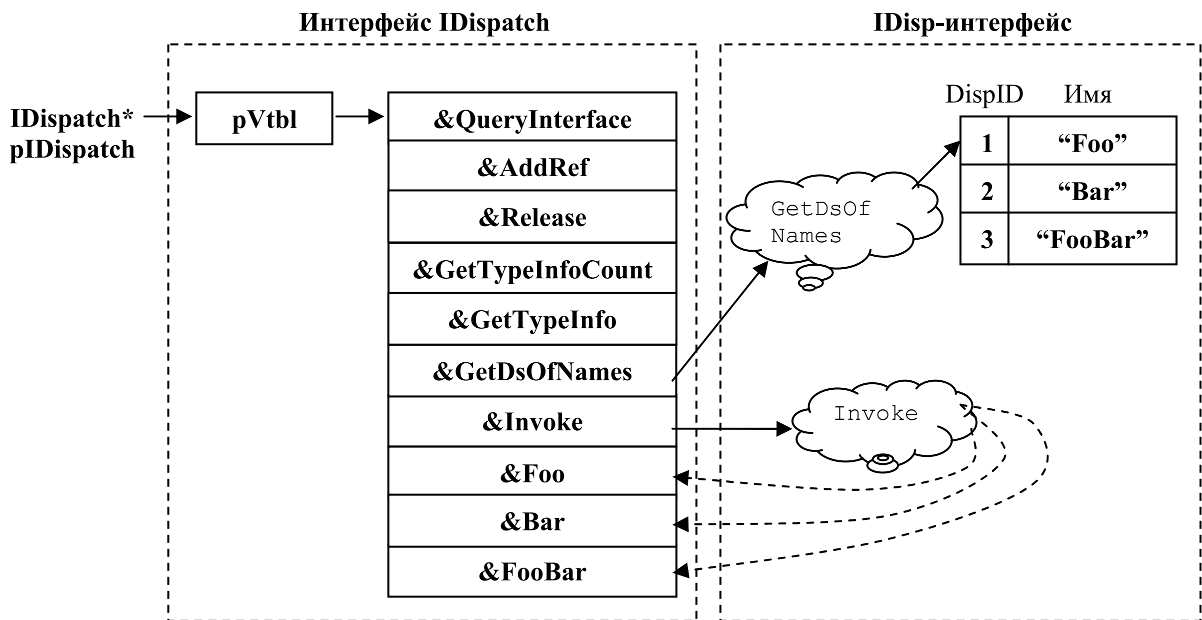


Рис. 12.3. *Дуальный интерфейс — это интерфейс COM, который наследует IDispatch.*

Доступ к членам такого интерфейса возможен и через Invoke, и через vtbl.

Дуальные интерфейсы предпочтительны для реализации disp-интерфейсов. Они позволяют программистам на C++ работать через vtbl; такие вызовы не только легче реализовать на C++, но они и быстрее выполняются. Макро- и интерпретируемые языки также могут использовать сервисы компонентов, реализующих дуальные интерфейсы, применяя *Invoke* вместо вызова через vtbl. Программа на Visual Basic может работать с дуальным интерфейсом как с disp-интерфейсом, так и через vtbl. Если Вы объявили тип переменной Visual Basic как *Object*, то работа идет через disp-интерфейс:

```
Dim doc As Object
Set doc = Application.ActiveDocument
doc.Activate
```

Если переменная имеет тип конкретного объекта, то Visual Basic выполняет вызов через vtbl:

```
Dim doc As Document
Set doc = Application.ActiveDocument
Doc.Activate
```

Однако если что-то выглядит слишком хорошо, чтобы быть правдой, — вероятно, так оно и есть. Наверное, Вы удивитесь, узнав, что у дуальных интерфейсов есть недостатки. С точки зрения Visual Basic, их и нет. Но с точки зрения контроллера Автоматизации, написанного на C++, их несколько. Основной из них — ограничения на типы параметров. Прежде чем обсудить ограниченность набора типов, допустимых для параметров disp-интерфейсов и дуальных интерфейсов, рассмотрим, как вызывается disp-интерфейс на C++.

Использование *IDispatch*

Рассмотрим следующую программу на Visual Basic:

```
Dim Cmpnt As Object
Set Cmpnt = CreateObject("InsideCOM.Chap11.Cmpnt11")
Cmpnt.Fx
```

Эта маленькая программа создает компонент COM и вызывает функцию *Fx* через интерфейс *IDispatch*, реализованный компонентом. Взглянем теперь на аналогичную программу на C++. Во-первых, необходимо создать компонент по его ProgID. (Эта процедура обсуждалась в гл. 6.) Приведенный ниже код, взятый из файла DCLIENT.CPP примера этой главы (который находится на прилагающемся к книге диске), создает компонент, используя ProgID. (Для ясности я убрал проверки ошибок.)

```
// Инициализировать библиотеку OLE.

HRESULT hr = OleInitialize(NULL);
// Получить CLSID приложения.
wchar_t progid[] = L"InsideCOM.Chap11";
CLSID clsid;
hr = ::CLSIDFromProgID(progid, &clsid);
// Создать компонент.
IDispatch* pIDispatch = NULL;
hr = ::CoCreateInstance(clsid, NULL, CLSCTX_INPROC_SERVER,
IID_IDispatch, (void**)&pIDispatch);
```

Чтобы не делать лишнего вызова *QueryInterface*, я запросил у *CoCreateInstance* указатель на *IDispatch*. Теперь, имея этот указатель, мы можем получить DISPID функции *Fx*. Функция *IDispatch::GetIDsOfNames* принимает имя функции и в виде строки возвращает соответствующий DISPID:

```
DISPID dispid;
OLECHAR* name = L"Fx";
pIDispatch->GetIDsOfNames(
            IID_NULL, // Должно быть IID_NULL
            &name, // Имя функции
            1, // Число имен
            GetUserDefaultLCID(), // Информация локализации
            &dispid); // Диспетчерский идентификатор
```

С точки зрения клиента DISPID — просто средство оптимизации, позволяющее избежать передачи строк. Для сервера же DISPID — идентификатор функции, которую хочет вызвать клиент. Имея DISPID для *Fx*, мы можем вызвать эту функцию, передав DISPID *IDispatch::Invoke*, которая представляет собой сложную функцию. Ниже приведен один из простейших вариантов вызова *Invoke*. Здесь *Fx* вызывается без параметров:

```
// Подготовить аргументы для Fx
DISPPARAMS dispparamsNoArgs = {
            NULL,
            NULL,
            0, // Ноль аргументов
            0, // Ноль именованных аргументов
};
```

```
// Простейший вызов Invoke
pIDispatch->Invoke(dispid, // DISPID
                   IID_NULL, // Должно быть IID_NULL
                   GetUserDefultLCID(), // Информация локализации
                   DISPATCH_METHOD, // Метод
                   &dispparamsNoArgs, // Аргументы метода
                   NULL, // Результаты
                   NULL, // Исключение
                   NULL); // Ошибка в аргументе
```

Контроллер Автоматизации не обязан что-либо знать о сервере Автоматизации. Контроллеру не нужен заголовочный файл с определением функции *Fx*. Информация об этой функции не зашита в программу. Сравните это с самим интерфейсом *IDispatch*, который является интерфейсом COM. *IDispatch* определен в OAIDL.IDL. код вызова членов *IDispatch* генерируется во время компиляции и остается неизменным. Однако вызываемая функция определяется параметрами *Invoke*. Эти параметры, как и параметры всех функций могут меняться во время выполнения.

Преобразовать приведенный выше фрагмент кода в программу, которая будет вызывать *любую* функцию без параметров, легко. Просто запросите у пользователя две строки — ProgID и имя функции — и передайте их *CLSIDFromProgID* и *GetIDsOfNames*. Код вызова *Invoke* останется неизменным.

Сила *Invoke* в том, что она может использоваться в полиморфно. Любой реализующий ее компонент можно вызывать при помощи одного и того же кода. Однако у этого есть своя цена. Одна из задач *IDispatch::Invoke* —передача параметров вызываемой функции. Число типов параметров, которые *Invoke* может передавать, ограничено. Более подробно об этом мы поговорим ниже, в разделе, где будет обсуждаться VARIANT. Но прежде чем поговорить о параметрах функций disp-интерфейсов, давайте рассмотрим параметры самой *IDispatch::Invoke*.

Параметры *Invoke*

Рассмотрим параметры функции *Invoke* более подробно. Первые три параметра объяснить нетрудно. Первый —это DISPID функции, которую хочет вызвать контроллер. Второй параметр зарезервирован и должен быть равен IID_NULL. Третий параметр содержит информацию локализации. Рассмотрим более детально оставшиеся параметры, начиная с четвертого.

Методы и свойства

Все члены интерфейса COM — функции. Интерфейсы COM, многие классы C++ и даже Win32 API моделируют доступ к переменной с помощью функций «Get» и «Set». Пусть, например, *SetVisible* делает окно видимым, а *GetVisible* возвращает текущее состояние видимости окна:

```
if (pIWindow->GetVisible() == FALSE)
{
  pIWindow->SetVsible(TRUE);
}
```

Но для Visual Basic функций «Get» и «Set» недостаточно. Основная задача Visual Basic — сделать все максимально простым для разработчика. Visual Basic поддерживает понятие *свойств (properties)*. Свойства — это функции «Get/Set», с которыми программист на Visual Basic работает как с переменными. Вместо синтаксиса вызова функции программист использует синтаксис обращения к переменной:

```
' Код VB
If Window.Visible = False Then
  Window.Visible = True
End If
```

Атрибуты IDL *propget* и *propput* указывают, что данная функция COM должна рассматриваться как свойство. Например:

```
[
  object,
  uuid(D15B6E20-0978-11D0-A6BB-0080C7B2D682),
  pointer_default(unique),
  dual
]
interface IWindow : IDispatch
{
  ...
  [propput]
  HRESULT Visible([in] VARIANT_BOOL bVisible);
  [propget]
  HRESULT Visible([out, retval] VARIANT_BOOL* pbVisible);
  ...
}
```

Здесь определяется интерфейс со свойством *Visible*. Функция, помеченная как *propput*, принимает значение свойства в качестве параметра. Функция помеченная *propget*, возвращает значение свойства как выходной параметр. Имена свойства и функции совпадают. Когда MIDL генерирует для функций *propget* и *propput* заголовочный файл, он присоединяет к имени функции префикс *get_* или *put_*. Следовательно, на C++ эти функции должны вызываться так:

```
VARIANT_BOOL vb;

get_Visible(&vb);
{
  put_Visible(VARIANT_TRUE);
}
```

Возможно, Вы уже начали понимать, почему я, программист на C++, не люблю disp-интерфейсы. Что хорошо на Visual Basic, плохо на C++. Я рекомендую Вам предоставлять интерфейс COM низкого уровня для пользователей Visual Basic и Java. Такой дуальный интерфейс можно реализовать с помощью интерфейсов COM низкого уровня. Писать хорошие интерфейсы достаточно сложно, даже если не пытаться удовлетворить два разных класса разработчиков. Кстати, VARIANT_TRUE — 0xFFFF. Возможно, Вы удивлены, какое все это имеет отношение к четвертому параметру *IDispatch::Invoke*. Все просто —одно имя, например *Visible*, может быть связано с четырьмя разными функциями: нормальной функцией, функцией установки значения свойства, функцией установки значения свойства по ссылке и функцией, которая возвращает значение свойства. У всех этих одноименных функций будет один и тот же DISPID, но реализованы они могут быть совершенно по-разному. Таким образом, *Invoke* необходимо знать, какую функцию вызывать.

Необходимая информация задается одним из следующих значенийчетвертого параметра:

```
DISPATCH_METHOD
DISPATCH_PROPERTGET
DISPATCH_PROPERTYPUT
```

```
DISPATCH_PROPERTYPUTREF
```

Параметры функций disp-интерфейсов

Довольно запутанно, не так ли? Пятый параметр IDispatch::Invoke содержит параметры вызываемой функции. Понятнее это будет на примере. Пусть мы вызываем *Invoke* для установки свойств *Visible* в *True*. Аргументы функции, к которой мы обращаемся, передаются в пятом параметре *Invoke*. Таким образом, нам нужно передать в этом пятом параметре *True* — новое значение свойства *Visible*. Пятый параметр — это структура DISPPARAMS, определение которой таково:

```
typedef struct tagDISPPARAMS {
            VARIANTARG* rgvarg; // Массив аргументов
            DISPID* rgdispidNamedArgs; // DISPID для именованных аргументов
            unsigned int cArgs; // Число аргументов
            unsigned int cNamedArgs; // Число именованных аргументов
} DISPPARAMS;
```

Visual Basic и disp-интерфейсы поддерживают концепцию *именованных аргументов*. Именованные аргументы позволяют программисту задавать параметры функции в любом порядке, передавая вместе со значениями параметра его имя. Эта концепция малополезна для программиста на C++, и у нас есть гораздо более важные
темы для обсуждения, поэтому я не собираюсь ее здесь рассматривать. В этой книге *rgdispidNamedArgs* всегда будет равен NULL, а *cNamedArgs* — 0.

Первый элемент *(rgvarg)* структуры DISPPARAMS — это массив аргументов. Поле *cArgs* задает число аргументов в данном массиве. Каждый аргумент имеет тип VARIANTARG, и именно поэтому число типов параметров, которые могут передаваться между контроллером и сервером Автоматизации, ограничено.

Функциям disp-интерфейса или дуального интерфейса можно передавать только такие параметры, которые можно поместить в структуру VARIANTARG (поскольку функции в vtbl должны соответствовать функциям, доступным через *Invoke*).
VARIANTARG — это то же самое, что и VARIANT. Знакомый нам файл IDL Автоматизации OAIDL.IDL дает следующее определение VARIANT:

```
typedef struct tagVARIANT {
      VARTYPE vt;
      unsigned short wReserved1;
      unsigned short wReserved2;
      unsigned short wReserved3;
      union {
            Byte              bVal;               // VT_UI1.
            Short             iVal;               // VT_I2.
            long              lVal;               // VT_I4.
            Float             fltVal;             // VT_R4.
            Double            dblVal;             // VT_R8.
            VARIANT_BOOL      boolVal;            // VT_BOOL.
            SCODE             scode;              // VT_ERROR.
            CY                cyVal;              // VT_CY.
            DATE              date;               // VT_DATE.
            BSTR              bstrVal;            // VT_BSTR.
            DECIMAL FAR*      pdecVal             // VT_BYREF|VT_DECIMAL.
            IUnknown FAR*     punkVal;            // VT_UNKNOWN.
            IDispatch FAR*    pdispVal;           // VT_DISPATCH.
            SAFEARRAY FAR*    parray;             // VT_ARRAY|*.
            Byte FAR*         pbVal;              // VT_BYREF|VT_UI1.
            short FAR*        piVal;              // VT_BYREF|VT_I2.
            long FAR*         plVal;              // VT_BYREF|VT_I4.
            float FAR*        pfltVal;            // VT_BYREF|VT_R4.
```

```
        double FAR*       pdblVal;          // VT_BYREF|VT_R8.
        VARIANT_BOOL FAR* pboolVal;         // VT_BYREF|VT_BOOL.
        SCODE FAR*        pscode;           // VT_BYREF|VT_ERROR.
        CY FAR*           pcyVal;           // VT_BYREF|VT_CY.
        DATE FAR*         pdate;            // VT_BYREF|VT_DATE.
        BSTR FAR*         pbstrVal;         // VT_BYREF|VT_BSTR.
        IUnknown FAR* FAR* ppunkVal;        // VT_BYREF|VT_UNKNOWN.
        IDispatch FAR* FAR* ppdispVal;      // VT_BYREF|VT_DISPATCH.
        SAFEARRAY FAR* FAR* pparray;        // VT_ARRAY|*.
        VARIANT FAR*      pvarVal;          // VT_BYREF|VT_VARIANT.
        void FAR*         byref;            // Generic ByRef.
        char              cVal;             // VT_I1.
        unsigned short    uiVal;            // VT_UI2.
        unsigned long     ulVal;            // VT_UI4.
        int               intVal;           // VT_INT.
        unsigned int      uintVal;          // VT_UINT.
        char FAR *        pcVal;            // VT_BYREF|VT_I1.
        unsigned short FAR * puiVal;        // VT_BYREF|VT_UI2.
        unsigned long FAR * pulVal;         // VT_BYREF|VT_UI4.
        int FAR *         pintVal;          // VT_BYREF|VT_INT.
        unsigned int FAR * puintVal;        //VT_BYREF|VT_UINT.
        };
    };
```

Как видите, VARIANT — это просто большое объединение (union) разных типов. VARIANT всегда использовался в Visual Basic для унифицированного хранения переменных разных типов. Идея оказалась настолько хороша, что разработчики Visual Basic решили выпустить ее в свет. Скоро мы рассмотрим, как ее использовать. Для нас, однако, важно то, что disp-интерфейсы и дуальные интерфейсы могут передавать только те типы, которые можно выразить при помощи VARIANT. Теперь продолжим рассмотрение *Invoke*.

Возврат результатов

Шестой параметр, *pVarResult* — это указатель на VARIANT, который будет содержать результат выполнения метода (или *propget*), исполняемого *Invoke*. Этот параметр может быть равен NULL для методов, не возвращающих значение, а также для *propput* и *propputref*.

Исключения

Следующий параметр *IDispatch::Invoke* — указатель на структуру EXCEPINFO. Если в процессе работы метода или свойства, вызванного с помощью *Invoke*, возникнет исключение (исключительная ситуация), структура будет заполнена информацией об этой ситуации. Структуры EXCEPINFO используются в тех же случаях, что и исключения в C++.

Ниже приведено определение EXCEPINFO. BSTR — это строка специального формата, о которой мы поговорим далее в этой главе.

```
typedef struct tagEXCEPINFO {
      WORD wCode;             // Код ошибки
      WORD wReserved;
      BSTR bstrSource;        // Источник исключительной ситуации
      BSTR bstrDescription;   // Описание ошибки
      BSTR bstrHelpFile;      // Полное имя файла справки
      DWORD dwHelpContext;    // Контекст внутри файла справки
      ULONG pvReserved;
```

```
    ULONG pfnDefferedFillIn; // Функция для заполнения этой структуры
    SCODE scode; // Код возврата
} EXCEPINFO;
```

Значение, идентифицирующее ошибку, должно содержаться либо в коде ошибки (*wCode*), либо в коде возврата (*scode*), при этом другое поле должно быть нулем. Ниже приведен простой пример использования структуры EXCEPINFO:

```
EXCEPINFO excepinfo;
HRESULT hr = pIDispatch->Invoke(..., &excepinfo);
if (FAILED(hr))
{
  // Ошибка при вызове Invoke
  if (hr == DISP_E_EXCEPTION)
  {
    // Метод сгенерировал исключение.
    // Сервер может отложить заполнение EXCEPTINFO.
    if (excepinfo.pfnDefferedFillIn != NULL)
    {
      // Заполнить структуру EXCEPTINFO
      (*(excepinfo.pfnDefferedFillIn)(&excepinfo);
    }
    strstream sout;

    sout << "Информация об исключительной ситуации в компоненте:"
         << endl
         << " Источник: " << excepinfo.bstrSource << endl
         << " Описание: " << excepinfo.bstrDescription
         << ends;
    trace(sout.str());
    }
  }
```

Ошибки в аргументах

Если возвращаемое значение *IDispatch::Invoke* равно либо DISP_E_PARAMNOTFOUND, либо DISP_E_TYPEMISMATCH, то индекс аргумента, вызвавшего ошибку, возвращается в последнем параметре — *puArgErr*.

Теперь, познакомившись со всеми параметрами *Invoke*, давайте рассмотрим еще один пример вызова функции disp-интерфейса. Затем более подробно поговорим о VARIANT, а также рассмотрим два типа, которые могут содержаться в VARIANT: BSTR и SAFEARRAY.

Примеры

Код примера этой главы содержит компонент, который реализует дуальный интерфейс *IX*. Весь код можно найти на прилагающемся к книге диске. Для компиляции при помощи Microsoft Visual C++ воспользуйтесь командой:

```
nmake –f makefile
```

По этой команде будут построены версии компонента внутри и вне процесса.

Интерфейс *IX* описан в SERVER.IDL так:

```
// Interface IX
[
  object,
```

```
  uuid(32BB8326-B41B-11CF-A6BB-0080C7B2D682),
  helpstring("Интерфейс IX"),
  pointer_default(unique),
  dual,
  oleautomation
]
interface IX : IDispatch
{
  import "oaidl.idl";
  HRESULT Fx();
  HRESULT FxStringIn([in] BSTR bstrIn);
  HRESULT FxStringOut([out, retval] BSTR* pbstrOut);
  HRESULT FxFakeError();
};
```

С этим компонентом могут работать два клиента. Клиент, содержащийся в файле CLIENT.CPP, подключается к компоненту с помощью vtbl, как мы делали в предыдущих главах. Клиент же из файла DCLIENT.CPP работает через disp-интерфейс. Ранее в этой главе мы уже видели, как он вызывает функцию *Fx*. Теперь посмотрим на вызов функции *FxStringIn*. Для большей ясности я убрал из кода обработку ошибок:

```
trace("Получить DispID метода \"FxStringIn\".");

name = L"FxStringIn";
hr = pIDispatch->GetIDsOfNames(IID_NULL,
                        &name,
                        1,
                        GetUserDefaultLCID(),
                        &dispid);

// Передать компоненту следующую строку
wchar_t wszIn[] = L"Это тестовая строка";

// Преобразовать строку Unicode в BSTR
BSTR bstrIn;
bstrIn = ::SysAllocString(wszIn);

// Подготовить параметры и осуществить вызов
// Выделить и инициализировать аргумент VARIANT
VARIANTARG varg;

::VariantInit(&varg); // Инициализировать VARIANT.
varg.vt = VT_BSTR; // Тип данных VARIANT
varg.bstrVal = bstrIn; // Данные для VARIANT

// Заполнить структуру DISPPARAMS
DISPPARAMS param;
param.cArgs = 1; // Один аргумент
param.rgvarg = &varg; // Указатель на аргумент
param.cNamedArgs = 0; // Нет именованных аргументов
param.rgdispidNamedArgs = NULL;
trace("Вызвать метод \"FxStringIn\".");

hr = pIDispatch->Invoke(dispid,
                  IID_NULL,
                  GetUserDefaultLCID(),
                  DISPATCH_METHOD,
                  &param,
                  NULL,
                  NULL,
```

```
                    NULL);

// Очистка
::SysFreeString(bstrIn);
```

На заполнение структур VARIANTARG и DISPPARAMS может уйти много строк. К счастью, Вы можете написать вспомогательные функции, которые значительно упростят вызов *Invoke*. Некоторые подобные функции можно найти внутри MFC. Кроме того, ClassWizard генерирует для disp-интерфейсов класс-оболочку C++. Подобные классы содержат удобные для работы на C++ функции, которые преобразуют свои параметры в формат, необходимый для вызова *Invoke*. Давайте воспользуемся случаем более подробно рассмотреть тип VARIANT. При рассмотрении типов мы также кратко познакомимся с типами BSTR и SAFEARRAY.

Тип VARIANT

Мы уже видели, как выглядит структура VARIANT (или VARIANTARG). Теперь давайте несколько подробнее рассмотрим, как она используется. Как видно из предыдущего фрагмента кода, структура VARIANT инициализируется при помощи *VariantInit*. Эта функция устанавливает поле *vt* в VT_EMPTY. После вызова *VariantInit* поле *vt* используется для указания типа данных, хранящихся в объединении VARIANT. В предыдущем примере мы сохраняли BSTR и поэтому использовали поле *bstrVal*.

Позднее связывание

При использовании класса C++ или интерфейса COM все параметры функций класса или интерфейса описываются в заголовочном файле. На этапе компиляции компилятор проверяет, чтобы каждой функции передавались параметры надлежащих типов. Строгая типизация — важное средство повышения надежности программ. Однако, это средство может оказаться слишком сильным, если Вы хотите написать простой макрос, где гибкость и простота важнее надежности.

Возможно, Вы не обратили внимания, но мы не предоставляли Visual Basic ничего, эквивалентного заголовочному файлу C++. Для того, чтобы разрешить программе вызвать метод диспетчерского интерфейса, Visual Basic не требуется знание аргументов этого метода. Достигается это при помощи структуры VARIANT.

Пусть в программе на Visual Basic имеется следующий фрагмент:

```
Dim Bullwinkle As Object
Set Bullwinkle = CreateObject("TalkingMoose")
Bullwinkle.PullFromHat 1, "Topolino"
```

Visual Basic не требуется знать о *Bullwinkle* ничего, кроме того, что тот поддерживает *IDispatch*. Поскольку же *Bullwinkle* поддерживает *IDispatch*, Visual Basic может получить DISPID для *PullFromHat* с помощью вызова *IDispatch::GetIDsOfNames*. Но у него нет никакой информации об аргументах *PullFromHat*. Здесь можно прибегнуть к помощи *библиотеки типа*, независимого от языка программирования эквивалента аголовочного файла C++. Мы будем рассматривать библиотеки типа далее в этой главе.

Но на самом деле Visual Basic не требует, чтобы ему сообщили допустимые типы параметров (через заголовочный файл или некий его эквивалент). Он может взять аргументы, введенные пользователем, и «засунуть» их в VARIANT. В предыдущем примере Visual Basic может предположить, что тип первого параметра — *long*, а второго — BSTR. Затем созданные таким образом «варианты» передаются функции *Invoke*. Если типы параметров не совпадают, сервер Автоматизации возвратит ошибки, возможно, вместе с индексом неправильного параметра. Конечно, программисту в любом случае

необходима некая документация функций, чтобы знать, как их вызывать, но самой программе никакая информация о типе не требуется. Использование VARIANT позволяет практически полностью отказаться от статической проверки типов — за счет того, что компонент будет проверять их во время выполнения. Это более похоже на Smalltalk, где нет проверки типов, чем на строгую типизацию C++.

Откладывая проверку типов до момента выполнения программы, мы требуем от диспетчерских методов и свойств способности проверять типы получаемых аргументов. Диспетчерские методы и свойства должны проверять корректность типов, иначе при выполнении макроса в сервере Автоматизацииможет возникнуть фатальная ошибка — что абсолютно неприемлемо.

Преобразование типов

Если хорошие disp-интерфейсы возвращают код ошибки при получении параметров неадекватного типа, то очень хорошие выполняют преобразование типа полученные аргументов за программиста. Возьмем функцию *PullFromHat* из предыдущего фрагмента. Visual Basic мог предположить, что функция принимает *long* и BSTR.

Но может оказаться, что на самом деле функция принимает не *long*, а *double*. Disp-интерфейс должен уметь выполнять такое преобразование автоматически. Кроме того, disp-интерфейсы должны выполнять преобразование в BSTR и из BSTR. Например, если функция установки значения свойства, описанная в IDL так:

```
[propput] HRESULT Title([in] BSTR bstrTitle);
```

вызывается следующим кодом на Visual Basic:

```
component.Title = 100
```

то эта функция должна преобразовать число 100 в BSTR и использовать результат преобразования в качестве заголовка (title). И, наоборот, функция установки значения свойства:

```
[propput] HRESULT Age([in] short sAge);
```

должна быть способна корректно выполнить следующий вызов из Visual Basic:

```
component.Age = “16”
```

Я не сомневаюсь, что у Вас нет никакого желания писать код этих преобразований. Даже если у Вас оно есть, то у других его нет. Хуже того, если преобразования будут писать все, все преобразования будут разными. В результате какие-то методы и свойства будут выполнять преобразования одним способом, а какие-то — другим.

Поэтому Автоматизация предоставляет функцию *VariantChangeType*, которая выполняет преобразование за Вас:

```
HRESULT VariantChangeType(
            VARIANTARG* pVarDest, // Преобразованное значение
            VARIANTARG* pVarSrc, // Исходное значение
            unsigned short wFlags,
            VARTYPE vtNew // Целевой тип преобразования
}
```

Пользоваться этой функцией очень легко. Например, приведенная ниже процедура преобразует VARIANT в *double* с помощью *VariantChangeType*:

```
BOOL VariantToDouble(VARIANTARG* pvarSrc, double dp)
{
  VARIANTARG varDest;
  VariantInit(&varDest);
  HRESULT hr = VariantChangeType(&varDest,
                  pvarSrc,
                  0, VT_R8);
  if (FAILED(hr))
  {
    return FALSE;
  }
  *pd = varDest.dblVal;
  return TRUE;
}
```

Необязательные аргументы

Метод disp-интерфейса может иметь необязательные аргументы. Если Вы не хотите задавать значение такого аргумента, просто передайте вместо него VARIANT с полем *vt*, установленным в VT_ERROR, и полем *scode*, равным DISP_E_PARAMNOTFOUND. В этом случае вызываемый метод должен использовать собственное значени по умолчанию.

Тип данных BSTR

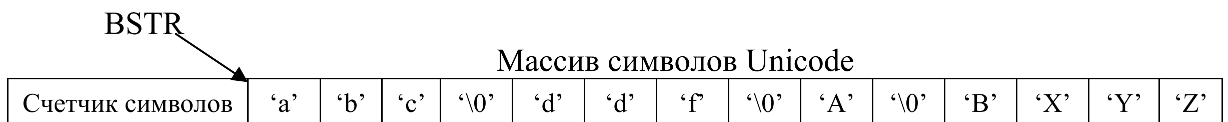
BSTR, сокращение от *Basic STRing* или *Binary STRing* (в зависимости от того, кого Вы спросите) — это указатель на строку символов Unicode. У BSTR есть три интересных особенности. Во-первых, в BSTR хранится число символов строки. Вторая важная особенность — то, что число хранится перед самим массивом символов (рис. 12.4).

Следовательно, нельзя объявить переменную типа BSTR и инициализировать ее массивом символов:

```
BSTR bstr = L"Где же счетчик?"; // Неправильно
```

Поскольку при этом не будет инициализирован счетчик. Вместо этого следует использовать функцию API Win32 *SysAllocString*:

```
wchar_t wsz[] = L"Вот где счетчик";
BSTR bstr;
bstr = SysAllocString(wsz);
```



Может содержать несколько нулевых символов

Рис. 12.4 *Счетчик символов хранится перед тем участком памяти, на который указывает BSTR*

По окончании использования BSTR следует освободить с помощью *SysFreeString*. Преобразовать BSTR обратно в строку *wchar_t* легко; в конце концов, BSTR указывает на начало массива *wchar_t*. Но у BSTR есть и третья интересная черта — в строке может содержаться несколько символов '\0'. Следовательно, Вы должны писать код, готовый к обработке нескольких символов '\0', если для Вашей функции это имеет смысл.

Тип данных SAFEARRAY

Другой специальный тип данных, который можно передавать disp-интерфейсу, — SAFEARRAY. Как следует из названия, это массив, содержащий информацию о своих границах. Ниже приведено объявление из OAIDL.IDL:

```
typedef struct tagSAFEARRAY {
      unsigned short cDims; // Число измерений
      unsigned short fFeatures;
      unsigned long cbElements; // Размер каждого элемента
      unsigned long clocks; // Счетчик блокировок
      BYTE* pvData; // Указатель на данные
      [size_is(cDims) SAFEARRAYBOUND rgsabound[];
} SAFEARRAY;

typedef struct tagSAFEARRAYBOUND {
      ULONG cElements; // Число элементов в данном измерении
      LONG lLBound; // Нижняя граница по данному измерению
} SAFEARRAYBOUND;
```

Поле *fFeatures* описывает, какого типа данные хранятся в SAFEARRAY. Возможны следующие значения:

FADF_BSTR Массив BSTR
FAFD_UNKNOWN Массив *IUnknown**
FADF_DISPATCH Массив *IDispatch**
FADF_VARIANT Массив VARIANT
Это поле также описывает, как массив был выделен:
FADF_AUTO Массив размещен в стеке
FADF_STATIC Массив размещен статически
FADF_EMBEDDED Массив входит в структуру
FADF_FIXEDSIZE Размер и местоположение массива нельзя менять

Библиотека Автоматизации — OLEAUT32.DLL — включает целый ряд функций для манипулирования SAFEARRAY. Названия всех таких функций начинаются с префикса *SafeArray*. Поищите их сами в диалоговой справочной системе.

Мы знаем, как заполнять переменные VARIANT, которые используются для построения структуры DISPPARAMS, которая передается *IDispatch::Invoke*, с помощью которой мы можем вызвать диспетчерские методы и получать доступ к диспетчерскимсвойствам. Теперь пришла пора кратко рассмотреть библиотеки типа — независимый от языка эквивалент заголовочных файлов C++.

Библиотеки типа

Как мы уже видели, программа на Visual Basic или C++ может управлять компонентом через disp-интерфейс, ничего не зная о типах, связанных с этим интерфейсом или его методами. Однако, если Вы можете засунуть горошину в ухо, это не означает, что так и следует поступать. Точно так же, если Вы *можете* писать программу на Visual Basic без информации о типах, это не означает, что так и надо делать.

Описанные в предыдущем разделе проверка и преобразование типов VARIANT на этапе выполнения требуют много времени и могут привести к скрытым ошибкам в программе. Программист может случайно перепутать местами два параметра в вызове функции, и компонент успешно преобразует их типы. Большое преимущество C++ перед C — более строгая проверка типов; она до некоторой степени обеспечивает уверенность в том, что программа работает, как предполагалось.

Нам нужен независимый от языка эквивалент заголовочных файлов C++, который подходил бы для интерпретируемых языков и сред макропрограммирования. Решение есть — *библиотека типа (type library)* COM, которая предоставляет информацию типа о компонентах, интерфейсах, методах, свойствах, аргументах и структурах. Содержимое библиотеки типа аналогично содержимому заголовочного файла C++. Библиотека типа — это откомпилированная версия файла IDL, к которой возможен доступ из программы.

Это не текст на каком-то языке, требующий синтаксического разбора, а двоичный файл. Библиотека Автоматизации предоставляет стандартные компоненты для создания и чтения таких двоичных файлов. Без библиотеки типа возможности Visual Basic работать с компонентами ограничены disp-интерфейсами. Если же библиотека типа имеется, Visual Basic может работать с компонентом напрямую через vtbl дуального интрфейса.

Доступ через vtbl быстрее, и он безопаснее с точки зрения приведения типа. Да, пока не забыл, — библиотека типа может также содержать строки справочной информации для всех содержащихся в ней компонентов, интерфейсов и функций. С помощью средства просмотра объектов, подобного имеющемуся в Visual Basic, программист может легко получить подсказку о любом свойстве или методе. Не правда ли, замечательно?

Создание библиотеки типа

Библиотеку типа создает функция *CreateTypeLib* из библиотеки Автоматизации. *CreateTypeLib* возвращает интерфейс *IcreateTypeLib*, который можно использовать для занесения в библиотеку различной информации.

Вряд ли Вам когда-нибудь потребуется использовать этот интерфейс; вместо него можно пользоваться IDL и компилятором MIDL. В гл. 10 мы использовали IDL и компилятор MIDL для генерации кода DLL заместителя/заглушки, но они подходят и для генерации библиотек типа.

ODL и *MkTypLib*

В «старые» времена компилятор MIDL нельзя было использовать для генерации библиотек типа. Вместо описания библиотек на IDL приходилось использовать другой язык — ODL. ODL компилировался в библиотеку типа с помощью программы *MkTypLib*. ODL был похож на IDL, но отличий было достаточно, чтобы затруднить их совместное использование. Поддержка двух файлов, содержащих одну и ту же информацию, — также напрасный расход времени. К счастью, IDL и MIDL при разработке Windows NT 4.0 были расширены для поддержки создания библиотек типа. Теперь ODL и *MkTypLib* стали не нужны и более не используются.

Оператор *library*

Основа создания библиотеки типа при помощи IDL — оператор *library*. Все, что находится внутри блока кода, ограниченного фигурными скобками, которые следуют за ключевым словом *library*, будет компилироваться в библиотеку типа. Файл IDL из примера гл. 12 показан в листинге 12.1. как видите, у библиотеки типа есть свои GUID, версия и *helpstring*.

SERVER.IDL

```
//
// Server.idl - Исходный файл IDL для Server.dll
//
// Этот файл будет обрабатываться компилятором MIDL для
// генерации библиотеки типа (Server.tlb) кода маршалинга.
//

// Интерфейс IX
[
  object,
  uuid(32BB8326-B41B-11CF-A6BB-0080C7B2D682),
  helpstring("Интерфейс IX"),
  pointer_default(unique),
  dual,
  oleautomation
]

interface IX : IDispatch
{
  import "oaidl.idl";
  HRESULT Fx();
  HRESULT FxStringIn([in] BSTR bstrIn);
  HRESULT FxStringOut([out, retval] BSTR* pbstrOut);
  HRESULT FxFakeError();
};

//
// Описание компонента и библиотеки типа
//
[
  uuid(D3011EE1-B997-11CF-A6BB-0080C7B2D682),
  version(1.0),
  helpstring("Основы COM, Глава 12 1.0 Библиотека типа")
]

library ServerLib
{
  importlib("stdole32.tlb");

  // Компонент
  [
    uuid(0C092C2C-882C-11CF-A6BB-0080C7B2D682),
    helpstring("Класс компонента")
  ]
  coclass Component
  {
    [default] interface IX;
  };
};
```

Листинг 12.1 *Файл IDL, используемый для генерации библиотеки SERVER.TLB*

Оператор *coclass* определяет компонент; в данном случае это *Component* с единственным интерфейсом *IX*. Компилятор MIDL сгенерирует библиотеку типа, содержащую *Component* и *IX*. *Component* добавляется к библиотеке типа, так как оператор *coclass* находится внутри оператора *library*. Интерфейс *IX* включается в библиотеку потому, что на него есть ссылка внутри оператора *library*.

Когда компилятор MIDL встречает в файле IDL оператор *library*, он автоматически генерирует библиотеку типа. В гл. 11 Вы видели, что компилятор MIDL генерировал библиотеку типа SERVER.TLB, даже когда она не была нам нужна.

Распространение библиотек типа

После генерации библиотеки типа Вы можете либо поставлять ее в виде отдельного файла, либо включить ее в Ваш EXE или DLL как ресурс. Большинство разработчиков предпочитает второй вариант, поскольку он упрощает установку приложения.

Использование библиотек типа

Первый шаг при использовании библиотек типа — ее загрузка. Для этого имеется несколько функций. Первая, которую следует попробовать, — *LoadRegTypeLib*, пытающаяся загрузить библиотеку по информации из Реестра Windows. Если эта функция потерпела неудачу, Вам следует использовать *LoadTypeLib*, которая загружает библиотеку с диска по имени файла, либо *LoadTypeLibFromResource*, которая загружает библиотеку типа из ресурса в EXE или DLL. *LoadTypeLib* должна в процесса загрузки регистрировать для Вас библиотеку типа. Однако если ей задано имя полного пути, библиотека зарегистрирована не будет (см. PSS ID Number Q131055).

Следовательно, после успешного вызова *LoadTypeLib* стоит вызвать *RegisterTypeLib*. Соответствующий код приведен в листинге 12.2.

Модифицированный код инициализации компонента из CMPNT.CPP

```
HRESULT CA::Init()
{
  HRESULT hr;

  // Динамически загрузить TypeInfo, если он еще не загружен
  if (m_pITypeInfo == NULL)
  {
    ITypeLib* pITypeLib = NULL;
    hr = ::LoadRegTypeLib(LIBID_ServerLib,
                  1, 0, // Номера версии
                  0x00,
                  &pITypeLib);
    if (FAILED(hr))
    {
      // Загрузить и зарегистрировать библиотеку типа
      hr = ::LoadTypeLib(wszTypeLibFullName, &pITypeLib);
      if(FAILED(hr))
      {
        trace("Вызов LoadTypeLib неудачен", hr);
        return hr;
      }

      // Убедиться, что библиотека типа зарегистрирована
      hr = RegisterTypeLib(pITypeLib, wszTypeLibFullName, NULL);
      if(FAILED(hr))
      {
        trace("Вызов RegisterTypeLib неудачен", hr);
        return hr;
      }
    }
```

```
    // Получить информацию типа для интерфейса объекта
    hr = pITypeLib->GetTypeInfoOfGuid(IID_IX, &m_pITypeInfo);
    pITypeLib->Release();
    if (FAILED(hr))
    {
      trace("Вызов GetTypeInfoOfGuid неудачен", hr);
      return hr;
    }
  }
  return S_OK;
}
```

Листинг 12.2 *Загрузка, регистрация и использование библиотеки типа*

После загрузки библиотеку можно использовать. *LoadTypeLib* и другие функции возвращают указатель на интерфейс *ItypeLib*, который используется для доступа к библиотеке типа. Обычно от библиотеки типа Вам требуется информация об интерфейсе или компоненте. Чтобы ее получить, функции *ITypeLib::GetTypeInfo* передается CLSID или IID, и она возвращает указатель на *ItypeInfo* для запрошенного элемента.

С помощью указателя *ItypeInfo* можно получить практически любую необходимую информацию о компонентах, интерфейсах, методах, свойствах, структурах и т.п. Но на самом деле большинство программистов на C++ никогда этим не пользуются — кроме, как Вы увидите в следующем разделе, тех случаев, когда нужно реализовать *IDispatch*. *ITypeInfo* может реализовать *IDispatch* автоматически.

Наиболее часто эти интерфейсы используются инструментами просмотра библиотек типа, т.е. программами, показывающими программисту содержимое библиотеки. Такой инструмент имеется в Visual Basic, он называется *Object Browser*. С его помощью Вы можете найти конкретный метод данного интерфейса и получить о нем справку.

Программа OleView — тоже средство просмотра библиотеки типа. Одна из замечательных возможностей OleView — способность создавать по информации библиотеки типа файл, похожий на файл IDL/ODL. Эта возможность очень полезна.

Библиотеки типа в Реестре

Мне по-настоящему нравятся компоненты, которые сами регистрируют себя в Реестре. Я испытываю огромное отвращение к написанию кода, помещающего данные в Реестр. К счастью, библиотеки типа регистрируются сами. Любопытным может быть интересно, какую именно информацию помещает в Реестр библиотека типа.

Запустите REGEDIT.EXE и откройте раздел *HKEY_CLASSES_ROOT\TypeLib*. Здесь Вы увидите множество LIBID, которые представляют собой GUID, идентифицирующие библиотеки типа. Откройте один из таких GUID и Вы найдете информацию, похожую на ту, что представлена на рис. 12.5.

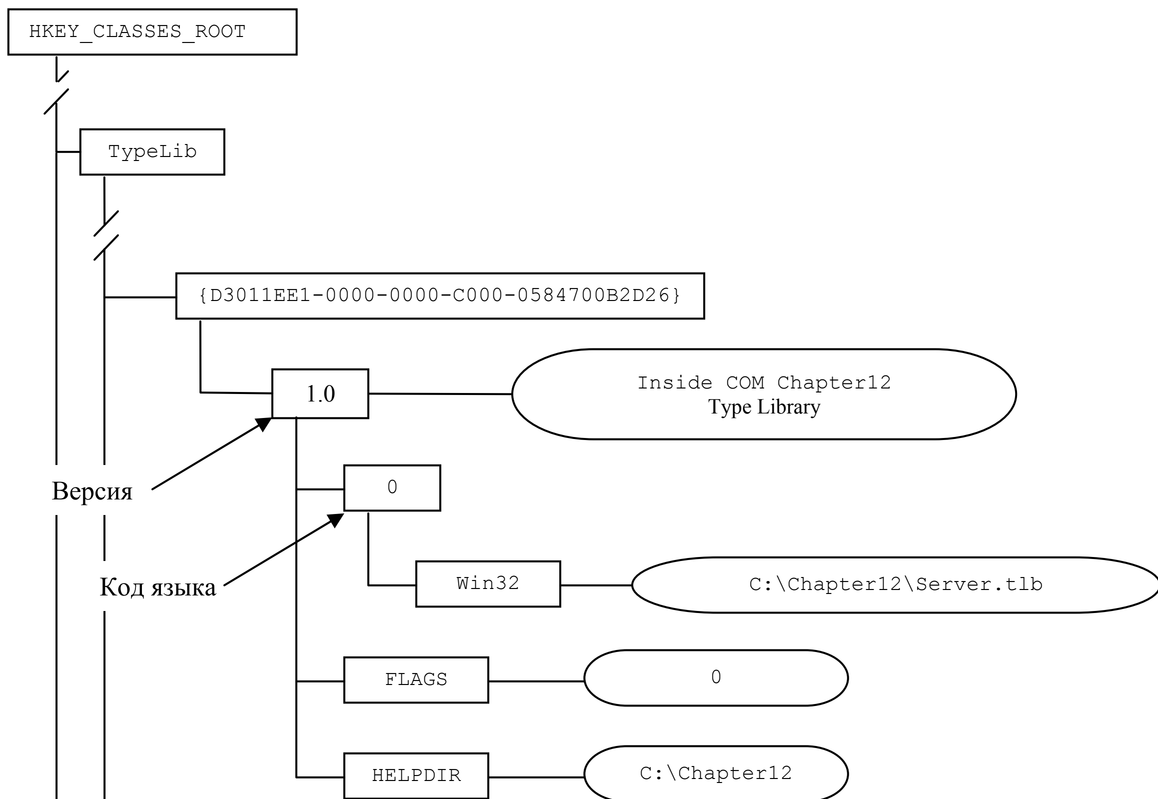


Рис. 12.5 *Информация, добавляемая в Реестр библиотекой типа*

Но одну вещь библиотеки типа не регистрируют: Вашему компоненту нужен в Реестре указатель на информацию его библиотеки типа. Поэтому Вы должны добавить раздел с именем *TypeLib* с GUID библиотеки в раздел CLSID Вашего компонента. Например, следующий раздел должен содержать указанный LIBID:

```
HKEY_CLASSES_ROOT\
                        CLSID\
                              {0C092C29-882C-11CF-A6BB-0080C7B2D682}\TypeLib
```

Библиотека типа создает раздел *TypeLib* для описанных в ней интерфейсов.

Как мы увидим в следующем разделе, библиотеки типа очень важны для реализации *IDispatch*.

Реализация *IDispatch*

Вероятно, способов реализации *IDispatch* не меньше, чем способов ободрать кошку. MFC сторит собственную таблицу имен и указателейна функции. Но реализация дуальных интерфейсов в MFC далека от элегантности. Я покажу Вам самый простой и популярный метод реализации *IDispatch*. В его основе лежит делегирование вызовов *GetIDsOfNames* и *Invoke* методам интерфейса *ITypeInfo*.

Я уже продемонстрировал Вам, как можно получить указатель *ITypeInfo* для интерфейса. Просто загрузите библиотеку типа и вызовите *ITypeLib::GetTypeInfoOfGuid*, передавая ей IID интерфейса. *GetTypeInfoOfGuid* возвращает указатель интерфейса *ITypeInfo*, который можно использовать для реализации *IDispatch*. Приведенный ниже код демонстрирует реализацию *IDispatch* (файл CMPNT.CPP из примера этой главы):

```
HRESULT __stdcall CA::GetTypeInfoCount(UINT* pCountTypeInfo)
{
  *pCountTypeInfo = 1;
  return S_OK;
}

HRESULT __stdcall CA::GetTypeInfo(
UINT iTypeInfo,
LCID, // Этот объект не поддерживает локализацию
ITypeInfo** ppITypeInfo)
{
  *ppITypeInfo = NULL;
  if(iTypeInfo != 0)
  {
    return DISP_E_BADINDEX ;
  }

  // Вызвать AddRef и вернуть указатель
  m_pITypeInfo->AddRef();
  *ppITypeInfo = m_pITypeInfo;
  return S_OK;
}

HRESULT __stdcall CA::GetIDsOfNames(
                        const IID& iid,
                        OLECHAR** arrayNames,
                        UINT countNames,
                        LCID, // Локализация не поддерживается
                        DISPID* arrayDispIDs)
{
  if (iid != IID_NULL)
  {
    return DISP_E_UNKNOWNINTERFACE;
  }
  HRESULT hr = m_pITypeInfo->GetIDsOfNames(arrayNames,
                                          countNames,
                                          arrayDispIDs);
  return hr;
}

HRESULT __stdcall CA::Invoke(
                        DISPID dispidMember,
                        const IID& iid,
                        LCID, // Локализация не поддерживается
                        WORD wFlags,
                        DISPPARAMS* pDispParams,
                        VARIANT* pvarResult,
                        EXCEPINFO* pExcepInfo,
                        UINT* pArgErr)
{
  if (iid != IID_NULL)
  {
    return DISP_E_UNKNOWNINTERFACE;
  }
  ::SetErrorInfo(0, NULL);
  HRESULT hr = m_pITypeInfo->Invoke(
  static_cast<IDispatch*>(this),
  dispidMember, wFlags, pDispParams,
  pvarResult, pExcepInfo, pArgErr);
  return hr;
}
```

Восхитительно просто, не так ли? У этого метода есть свои ограничения: например, он не поддерживает интернационализацию. К счастью, большинству компонентов это не нужно. Если Ваш компонент не таков, Вы можете загружать разные библиотеки типа на основании LCID, переданного в вызове *Invoke*.

Генерация исключений

Как упоминалось в разделе, посвященном параметрам *IDispatch::Invoke*, предпоследний параметр — структура EXCEPINFO. Чтобы заставить *ITypeInfo::Invoke* заполнить ее, Вы должны проделать следующую последовательность действий:

1. Реализуйте для своего компонента интерфейс *ISupportErrorInfo* с единственной функцией-членом:

```
// ISupportErrorInfo
virtual HRESULT __stdcall InterfaceSupportsErrorInfo(const IID& iid)
{
  return (iid == IID_IX) ? S_OK : S_FALSE;
}
```

2. В своей реализации *IDispatch::Invoke* вызовите *SetErrorInfo(0, NULL)* перед вызовом *ITypeInfo::Invoke*.

3. При возникновении исключительной ситуации вызовите *CreateErrorInfo*, чтобы получить указатель на интерфейс *ICreateErrorInfo*.

4. С помощью этого интерфейса предоставьте информацию об ошибке.

5. Наконец, получите указатель на интерфейс *IErrorInfo* и вызовите *SetErrorInfo*, передав ей в качестве второго параметра полученный указатель. Первый параметр зарезервирован и всегда равен 0. Все остальное — дело *ITypeInfo* и клиента.

Ниже приведен пример генерации исключения.

```
// Создать объект «Информация об ошибке»
ICreateErrorInfo* pICreateErr;
HRESULT hr = ::CreateErrorInfo(&pICreateErr);
if (FAILED(hr))
{
  return E_FAIL;
}

// pICreateErr->SetHelpFile(...);
// pICreateErr->SetHelpContext(...);
pICreateErr->SetSource(L"InsideCOM.Chap11");
pICreateErr->SetDescription(
L"Это фиктивная ошибка, сгенерированная компонентом");
IErrorInfo* pIErrorInfo = NULL;
hr = pICreateErr->QueryInterface(IID_IErrorInfo,
(void**)&pIErrorInfo);
if (SUCCEEDED(hr))
{
  ::SetErrorInfo(0L, pIErrorInfo);
  pIErrorInfo->Release();
}

pICreateErr->Release();
return E_FAIL;
```

Маршалинг

Если Вы посмотрите на make-файл из примера гл. 11, то увидите, что я не создаю DLL заместителя/заглушки. Это связано с тем, что система, а именно OLEAUT32.DLL, автоматически реализует маршалинг интерфейсов, совместимых с Автоматизацией1. Интерфейс, совместимый с Автоматизацией, наследует *IDispatch* и использует только такие типы параметров, которые можно поместить в VARIANT. Для таких типов OLEAUT32.DLL выполняет маршалинг автоматически.

Чтобы понять, как это работает, рассмотрим информацию в Реестре для версии интерфейса *IX* этой главы:

```
HKEY_CLASSES_ROOT\
        Interfaces\
            {32BB8326-B41B-11CF-A6BB-0080C7B2D682}\
                  ProxyStubClsid32
```

В этом разделе должен находиться следующий CLSID:

```
      {00020424-0000-0000-C000-000000000046}
```

Теперь найдем этот CLSID в разделе Реестра CLSID:

```
      HKEY_CLASSES_ROOT\
                  CLSID\
                  {00020424-0000-0000-C000-000000000046}\InprocServer32
```

Вы увидите, что значением *InprocServer32* является OLEAUT32.DLL

Что Вы хотите сделать сегодня?

Теперь Вы это получили: есть еще один способ коммуникации между клиентом и компонентом. Как обычно, если одно и то же можно сделать по-разному, Вы должны решить, что выбрать. Есть три варианта: интерфейсы vtbl, дуальные интерфейсы и disp-интерфейсы. Какой из них подойдет Вам? Как в таких случаях говорит мой отец: «С одной стороны шесть, с другой полдюжины». Есть, однако, вполне четкие рекомендации, какой тип интерфейса кода следует использовать.

Если доступ к Вашему компоненту будет осуществляться только из компилируемых языков типа C и C++, используйте vtbl или нормальный интерфейс COM. Интерфейсы vtbl работают значительно быстрее disp-интерфейсов. Кроме того, с ними гораздо легче работать на C++. Если к Вашему компоненту будут обращаться из Visual Basic или Java, следует реализовать дуальный интерфейс. Visual Basic и Java могут работать с ним либо как с disp-интерфейсом, либо через vtbl. На C++ также можно будет использовать оба эти способа.

Однако реализованная с помощью vtbl часть дуального интерфейса, который разработан специально для использования Visual Basic, вряд ли осчастливит большинство программистов на C++ (если только Вы не используете расширения компилятора Visual C++ 5.0). В связи с этим я рекомендую разработать низкоуровневый интерфейс vtbl и высокоуровневый дуальный интерфейс. Низкоуровневый интерфейс способен дать программисту на C++ дополнительную информацию, необходимую для эффективного агрегирования компонента.

Если только всерьез не нужно создавать компоненты во время выполнения, я бы вообще избегал реализации чистых disp-интерфейсов. Дуальные интерфейсы гораздо более гибки.

Кроме того, есть еще один фактор, влияющий на Ваше решение, — скорость. Если Вы имеете дело с компонентом внутри процесса, интерфейс vtbl работает примерно в 100 раз быстрее disp-интерфейса. (Точное значение несколько меняется в зависимости от набора типов аргументов функций.) В случае же компонента вне процесса накладные расходы маршалинга более существенны, чем накладные расходы *IDispatch::Invoke*, и интерфейс vtbl работает примерно лишь в два с половиной раза быстрее disp-интерфейса. Если же Ваш компонент является удаленным, то тип используемого интерфейса вообще не имеет значения

13. Многопоточность

Входящие в мой офис посетители постоянно бьются лбом о прикрепленный к потолку черный предмет сантиметров 30 длиной. Это копия вертолета Bell 206B-III Jet Ranger в масштабе 1:32. Копия не абсолютно точная — вместо хвостового винта сзади толкающий пропеллер благодаря которому модель может летать по кругу.

Выглядит это так. Вы включаете вертолет, и небольшой электромотор начинает вращать пропеллер. Мощности пропеллера не хватает, чтобы запустить движение, — надо слегка подтолкнуть. После этого вертолет начинает раскачиваться на подвесном шнуре. Постепенно пропеллер его разгоняет, угол между подвеской и потолком становится все меньше, и наконец, модель начинает быстро кружиться под потолком.

У этого маленького вертолета своя история. Его подарил мне Рёдигер Эш (Ruediger Asche), с которым мы вместе писали статьи для Microsoft Developer Network. Он знаток мрачных глубин ядра Windows NT, куда никогда не проникает свет GUI. Одна из областей специализации Рёдигера — многопоточное программирование. Вот мы и добрались до темы этой главы.

Если бы мы хотели написать программу моделирования моего вертолета, то могли бы использовать несколько потоков. Один из них отвечал бы за пользовательский интерфейс, дающий пользователю возможность управлять трехмерным изображением вращающегося вертолета. Другой бы вычислял положение вертолета при движении
по кругу и вверх.

Однако для моделирования летающего по кругу вертолета многопоточность необязательна. По-настоящему она бывает полезна при построении пользовательского интерфейса с малым временем отклика. Интерфейс можно сделать живее и «доступнее», если переложить вычисления на фоновый поток. Наиболее это заметно в программах просмотра Web. Большинство из них перекачивают страницу данных в рамках одного потока, а выводят на экран в рамках другого; третий же дает возможность пользователю работать со страницей во время ее загрузки. Лично я не выношу сидеть и ждать, пока загружается куча ненужных картинок, — так что обычно щелкаю мышью на следующей гиперссылке еще до окончания загрузки и отрисовки. Эта удобная возможность обеспечивается многопоточностью.

Поскольку потоки так важны для быстрого отклика приложений, есть основания ожидать, что доступ к компоненту СОМ будет осуществляться несколькими потоками. Однако с использованием компонента несколькими потоками связан ряд специфических проблем, которые мы рассмотрим в данной главе. Эти проблемы незначительны и несопоставимы по масштабу с более общей проблемой многопоточного программирования. Мы не будем подробно рассматривать многопоточное программирование; посмотрим лишь, как многопоточность влияет на разработку и использование компонентов СОМ. Более подробно о многопоточном программировании можно прочитать в статьях Рёдигера Эша в MSDN.

Потоковые модели СОМ

СОМ использует потоки Win32 и не вводит новых типов потоков или процессов. В СОМ нет своих примитивов синхронизации, для создания и синхронизации потоков просто используется API Win32. Использование потоков в СОМ, кроме некоторых нюансов, не отличается от их использованияв приложениях Win32. Мы рассмотрим эти
нюансы, но сначала позвольте мне привести общий обзор потоков Win32.

Потоки Win32

В обычном приложении Win32 имеются потоки двух типов: *потоки пользовательского интерфейса (userinterface threads)* и *рабочие потоки (worker threads)*. С потоком пользовательского интерфейса связаны одно или несколько окон. Такие потоки имеют циклы выборки сообщений, которые обеспечивают работу окон и реакцию на действия пользователя. Рабочие потоки используются для фоновой обработки и не связаны с окнами; в них обычно нет циклов выборки сообщений. В каждом процессе может быть несколько потоков пользовательского интерфейса и несколько рабочих потоков. У потоков пользовательского интерфейса есть интересная особенность поведения. Как я только что сказал, у каждого потока пользовательского интерфейса есть одно или несколько окон. Оконная процедура данного окна вызывается только потоком, который владеет этим окном, — т.е. потоком, создавшим окно. Таким образом, оконная процедура всегда выполняется в одном и том же потоке, независимо от того, какой поток послал сообщение этой процедуре на обработку. Следовательно, все посланные данному окну сообщения синхронизированы, и окно с гарантией будет получать сообщения упорядоченно.

Преимущества для Вас, программиста, — в том, что нет нужды писать «потокобезопасные» оконные процедуры (а их писать не просто и, возможно, небыстро). Поскольку синхронизацию сообщений гарантирует Windows, Вам не нужно беспокоиться о том, что оконную процедуру могут вызвать одновременно несколько потоков. Эта синхронизация весьма полезна потокам, управляющим пользовательским интерфейсом. В конце концов, мы хотим, чтобы информация о действиях пользователя достигала окна в той же последовательности, в какой эти действия производились.

Потоки COM

COM использует те же два типа потоков, хотя и называет их по-другому. Вместо «поток пользовательского интерфейса» в COM говорят *разделенный поток (apartment thread)*. Термин *свободный поток (free thread)* используют вместо термина «рабочий поток». Самая сложная часть потоковой модели COM — терминология.
Основная же сложность в ней состоит в несогласованности документации. Набор терминов, используемых в Win32 SDK, отличается от набора, используемого спецификацией COM. Я буду максимально избегать этой терминологии либо вводить термины как можно раньше. В этой главе я буду использовать термин *разделенный поток* для обозначения потока, подобного потоку пользовательского интерфейса, а термин *свободный поток* —для обозначения потока, подобного рабочему потоку.
Почему в COM вообще рассматривается потоковая модель, если она ничем не отличается от Win32? Причин две: маршалинг и синхронизация. Более подробно мы рассмотрим маршалинг и синхронизацию после того, как разберемся, что такое *подразделение (apartment)*, *модель разделенных потоков (apartment threading)* и *модель свободных потоков (free threading)*.

Подразделение

Хотя мне всерьез хотелось бы избежать новой терминологии, сейчас я определю термин *подразделение (apartment)*. Подразделение — это концептуальный конгломерат, состоящий из потока в стиле пользовательского интерфейса (так называемый разделенный поток) и цикла выборки сообщений.

Возьмем типичное приложение Win32, которое состоит из процесса, цикла выборки сообщений и оконной процедуры. У каждого процесса есть как минимум один поток. Схематически приложение Windows представлено на рис. 13.1. Рамка пунктирными

краями обозначает процесс. Рамка, внутри которой изображен цикл, представляет циклю выборки сообщений Windows. Две другие рамки изображают оконную процедуру и код программы. Все они расположены поверх линии, обозначающей поток управления. Кроме процесса, рис. 13.1 иллюстрирует и подразделение. Один поток — это разделенный поток. На рис. 13.2 та же схема иллюстрирует организацию типичного приложения COM, состоящего из клиента и двух компонентов внутри процесса. Программа работает внутри одного процесса и имеет единственный поток управления. У компонентов внутри процесса нет своих циклов выборки сообщений — они используют тот же цикл, что и клиентский EXE. И снова рисунок иллюстрирует одно подразделение.

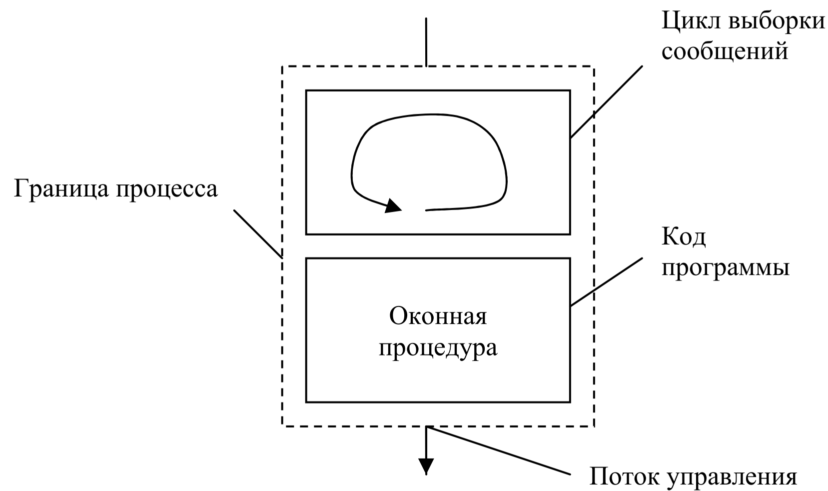


Рис. **13.1** *Приложение Windows. Показаны: поток управления, цикл выборки сообщений, граница процесса и код программы*

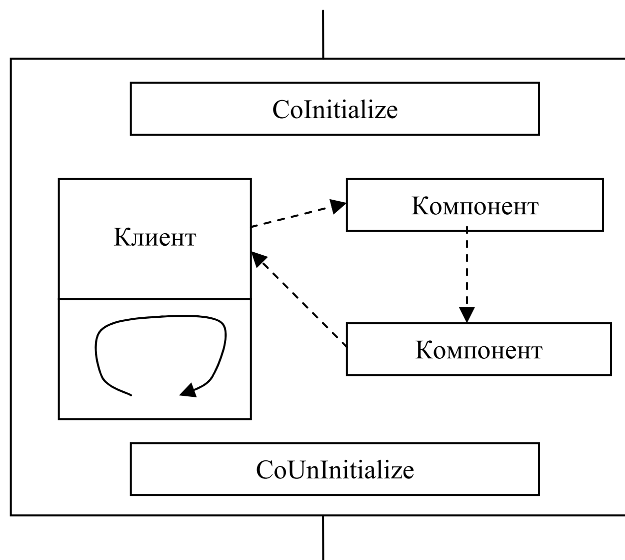


Рис. 13.2 *Клиент и два компонента внутри процесса. Имеется только один поток, и компоненты используют цикл выборки сообщений клиента совместно с клиентом*

Использование компонентов внутри процесса не изменяет базовой структуры приложения Windows. Самой существенное различие между двумя изображенными процессами — в том, что процесс с компонентами обязан, прежде чем использовать какие-либо функции библиотеки COM, вызвать *CoInitialize*, а перед завершением вызывать *CoUninitialize*.

Добавим компонент вне процесса

Когда клиент подсоединяется к компоненту вне процесса, картина меняется. Такой клиент показан на рис. 13.3. Компонент находится в процессе, отдельном от процесса клиента. У каждого процесса свой поток управления. Цикл выборки сообщений предоставляется компоненту его сервером вне процесса. Если вернуться к примеру гл. 11 код такого цикла можно найти в OUTPROC.CPP. Другое существенное отличие от случая с компонентом
внутри процесса — необходимость маршалинга вызовов между процессами. На рисунке такой вызов представлен«молнией». В гл. 11 узнали, как создать DLL заместителя/заглушки, которая используется для маршалинга данных между клиентом и компонентом вне процесса.

На рис. 13.3 изображены два подразделения. В одном из них находится клиент, а в другом — компонент. Может показаться, что подразделение — то же самое, что и процесс, но это неверно. В одном процессе может быть несколько подразделений.

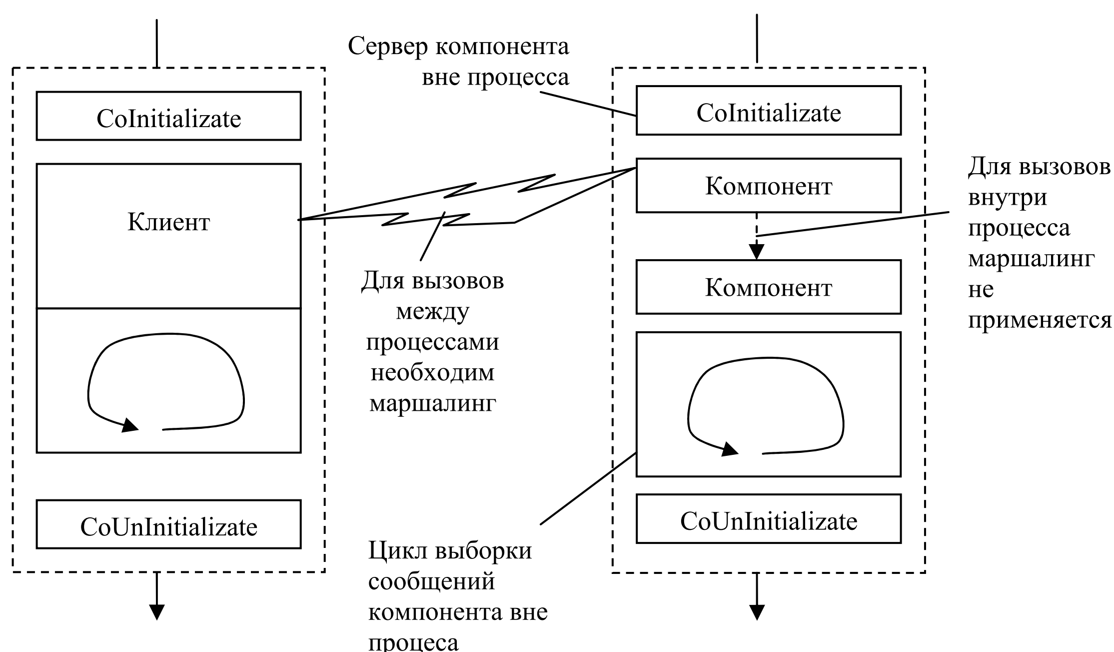


Рис. 13.3 *У компонента вне процесса есть собственный цикл выборки сообщений и поток*

На рис. 13.4 я превратил компонент рис. 13.3 из компонента вне процесса в компонент внутри процесса, расположенный в другом подразделении.

Штриховыми линиями изображены подразделения. Пунктирная линияпо-прежнему обозначает границу процесса. Обратите внимание, как похожи два рисунка. По существу, я нарисовал вокруг старой картинки новую рамку и объявил, что теперь объекты находятся в одном процессе. Из этого должна стать очевидной моя точка зрения —
подразделения аналогичны (однопоточным) процессам в следующих моментах. И у процесса, и у подразделения есть собственный цикл выборки сообщений. Маршалинг вызовов функций внутри (однопоточного) процесса и внутри подразделения не нужен. Имеет место естественная синхронизация, так как и у процесса, и у подразделения только один поток. Синхронизация вызовов функций между процессами и между подразделениями производится при помощи цикла выборки сообщений. И последняя деталь — каждый процесс должен инициализировать библиотеку COM. Точно так же и каждое подразделение должно инициализировать библиотеку COM. Теперь, если вернуться к рис. 13.2, Вам станет понятно, как клиент и два компонента сосуществуют внутри одного подразделения.

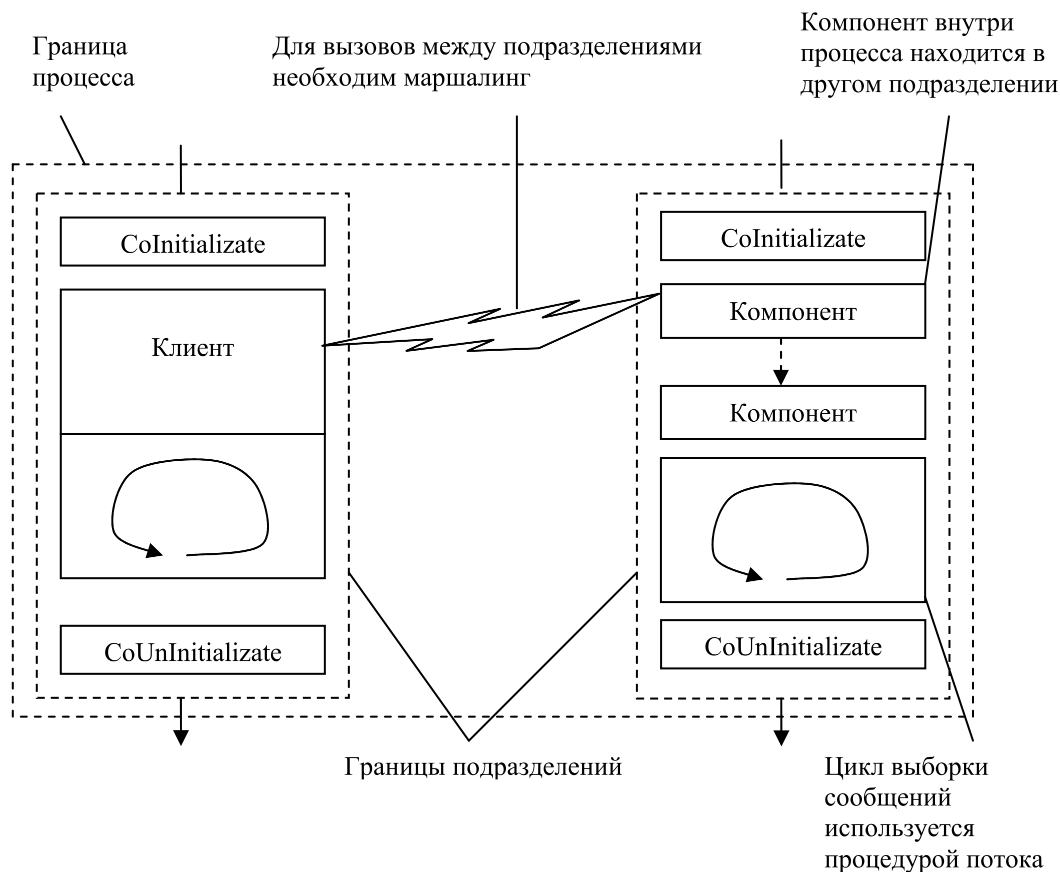


Рис. 13.4 *Клиент взаимодействует внутри процесса с компонентом, расположенным в другом подразделении*

Разделенные потоки

Разделенный поток — это единственный поток внутри подразделения. Но когда Вы слышите термин «разделенный поток», представляйте себе поток пользовательского интерфейса. Вспомните, что поток пользовательского интерфейса владеет созданным им окном. Оконная процедура вызывается только им. Между разделенным потоком и созданным им компонентом существуют такие же отношения. Разделенный поток
владеет созданным им компонентом. Компонент внутри подразделения будет вызываться только соответствующим разделенным потоком.

Если поток посылает сообщение окну, принадлежащему другому потоку, Windows помещает это сообщение в очередь сообщений соответствующего окна. Цикл выборки сообщений этогоокна выполняется потоком, создавшим окно. Когда цикл выбирает очередное сообщение и вызывает оконную процедуру, для вызова процедуры используется тот же поток, который создал окно.

То же самое верно для компонента внутри подразделения. Предположим, что метод Вашего компонента, находящегося в подразделении, вызывается другим потоком. COM автоматически помещает этот вызов в очередь подразделения. Цикл выборки сообщений извлекает этот вызов и вызывает метод с помощью потока подразделения.

Таким образом, компонент внутри подразделения вызывается только потоком подразделения, и ему нет нужды заботиться о синхронизации. Так как COM гарантирует, что все вызовы такого компонента будут упорядочены, компоненту не требуется быть «потокобезопасным». Это значительно облегчает написание кода компонента. Ни один из компонентов, которые мы написали в этой книге, не был «потокобезопасным». Но, пока их создают разделенный потоки, мы можем быть уверены, что их методы никогда не

будут вызваны разными потоками одновременно. Именно в этом состоит отличие свободных потоков от разделенных.

Свободные потоки

СОМ упорядочивает вызовы компонентов для разделенных потоков. Однако синхронизация не выполняется для компонентов, созданных свободными потоками. Если компонент создан свободным потоком, он может вызываться любым потоком и в любой момент времени. Разработчик должен гарантировать, что его компонент сам синхронизирует доступ к себе. Такой компонент должен быть «потокобезопасным». Модель свободных потоков переносит заботу о синхронизации с СОМ на компонент.

Поскольку СОМ не выполняет синхронизацию вызовов компонентов, свободным потокам не нужен цикл выборки сообщений. Компонент, созданный свободным потоком, называется компонентом свободных потоков. Такой компонент не принадлежит создавшему его потоку, а используется всеми потоками совместно: все потоки
имеют к нему свободный доступ. Разделенные потоки — единственный тип потоков, которые можно использовать при работе СОМ в Microsoft Windows NT 3.51 и Microsoft Windows 95. В Windows NT 4.0 и в Windows 95 с установленной поддержкой DCOM
можно использовать свободные потоки.

Мы познакомились со свободными потоками в общем. С более интересными подробностями мы столкнемся при обсуждении маршалинга и синхронизации.

Маршалинг и синхронизация

Для правильного маршалинга и синхронизации вызовов компонента СОМ надо знать, потоком какого типа он исполняется. В случае разделенных потоков СОМ обычно выполняет необходимые маршалинг и синхронизацию.

Для свободных потоков маршалинг может быть не нужен, а синхронизация возлагается на компонент. Запомните следующие общие правила:

- Вызовы между процессами всегда выполняются с использованием маршалинга. Мы обсуждали это в гл. 11.
- Вызовы внутри одного потока никогда не используют маршалинг.
- Вызов компонента в разделенном потоке выполняется с маршалингом.
- Вызов компонента в свободном потоке не всегда использует маршалинг.
- Вызовы с помощью разделенного потока синхронизируются.
- Вызовы с помощью свободного потока не синхронизируются.
- Вызовы внутри потока синхронизируются самим потоком.

Теперь рассмотрим возможные комбинации вызовов разделенных и свободных потоков. Давайте начнем с простых случаев. Если явно не указано иное, подразумевается, что вызовы осуществляются в пределах одного процесса.

Вызовы внутри одного потока

Если клиент, выполняющийся в каком-либо потоке, вызывает компонент, выполняющийся в том же потоке, то вызов синхронизирован просто потому, что поток всего один. СОМ не нужно выполнять какую-либо синхронизацию, и компонент не должен быть «потокобезопасным». Вызовы в пределах одного потока не требуют маршалинга. Это правило мы использовали на протяжении всей книги.

Разделенный — разделенный

Если клиент, выполняющийся в разделенном потоке, вызывает компонент, выполняющийся в другом разделенном потоке, то синхронизацию вызова выполняет COM. COM также выполняет маршалинг интерфейсов, даже если оба потока находятся в одном процессе. В некоторых случаях требуется выполнить маршалинг интерфейса между разделенными потоками вручную. Мы рассмотрим этот случай позже, когда будем реализовывать разделенныйпоток. Вызов компонента в разделенном потоке аналогичен вызову компонента вне процесса.

Свободный — свободный

Если клиент, выполняющийся в свободном потоке, вызывает компонент свободных потоков, то COM не будет синхронизировать этот вызов. Вызов будет выполнять поток клиента. Компонент должен сам синхронизировать доступ к себе, так как одновременно его может вызвать другой клиент посредством другого потока. Если клиент и компонент находятся внутри одного процесса, маршалинг вызова не выполняется.

Свободный — разделенный

Если клиент в свободном потоке вызывает компонент в подразделении, то синхронизацию вызова осуществляет COM. Компонент будет вызван потоком подразделения. Маршалинг интерфейса также необходим, независимо от того, в одном ли процессе или в разных находятся оба потока. В большинстве случаев маршалинг за Вас выполнит COM. Но иногда, как Вы скоро увидите, маршалинг приходится выполнять вручную.

Разделенный — свободный

Если клиент в разделенном потоке вызывает свободный поток, то COM не выполняет синхронизацию вызова. Эта обязанность лежит на компоненте свободного потока. Маршалинг интерфейса выполняется, но, если оба потока принадлежат одному процессу, COM может оптимизировать маршалинг, чтобы передавать указатели клиенту непосредственно. Подробнее мы рассмотрим это при реализации свободных потоков.
Как Вы видите, потоковые модели компонентов COM не слишком отличаются от обычных моделей потоков Win32. В процессе может быть любое число потоков. Эти потоки могут быть разделенными или свободными. С точки зрения программиста, в модели потоков COM есть только два интересных момента: синхронизация и маршалинг. COM синхронизирует вызовы компонентов в разделенных потоках. Разработчик синхронизирует вызовы компонентов в свободных потоках. Синхронизация компонентов в свободных потоках — это общая проблема многопоточности, а не специфика COM. Однако маршалинг специфичен для COM — и это единственное, что действительно уникально при работе с компонентами COM в многопоточной среде. Подробно мы рассмотрим ручной маршалинг интерфейсов позже, когда реализуем разделенный поток и поток клиента.

Реализация модели разделенных потоков

С компонентами внутри подразделений хорошо то, что им нет необходимости быть «потокобезопасными». Доступ к ним синхронизируется COM. При этом ни имеет значения, приходит ли вызов из потоков других подразделений или из свободных потоков. COM автоматически использует скрытую очередь сообщений Windows для

синхронизации клиентских вызовов таких компонентов. Благодаря этому реализация компонентов в однопоточных подразделениях очень схожа с написанием оконных процедур. (Для синхронизации доступа к оконной процедуре используется цикл выборки сообщений; COM использует тот же механизм для синхронизации доступа к однопоточному подразделению.) Ниже следуют основные требования к подразделению:

- Оно должно вызывать *CoInitialize* или *OleInitialize*.
- В нем может быть только один поток.
- У него должен быть цикл выборки сообщений.
- Оно обязано выполнять маршалинг указателей на интерфейсы при передаче их другим подразделениям.
- В случае компонента внутри процесса подразделение должно иметь «потокобезопасные» точки входа DLL.
- Ему может понадобиться «потокобезопасная» фабрика класса.

В следующих параграфах некоторые из этих требований рассматриваются подробнее.

Компонент может существовать только в одном потоке

Компонент в однопоточном подразделении должен выполняться в единственном потоке. Доступ к компоненту имеет только создавший его поток. Именно та работает оконная процедура — ее вызывает только поток, создавший данное окно. Так как доступ к компоненту возможен только из одного потока, такой компонент всегда выполняется в однопоточном подразделении и ему не нужно заботиться о синхронизации. Однако компонент должен защищать свои глобальные данные, поскольку он, как и оконная процедура, доступен для повторного входа (реентерабелен).

Необходим маршалинг интерфейсов через границы подразделений

Для вызовов из других потоков необходим маршалинг — чтобы вызовы выполнял тот же поток, в котором выполняется компонент. Так как в подразделениитолько один поток, все остальные потоки находятся вне подразделения. При вызове через границы подразделений всегда требуется маршалинг. Несколько более подробно маршалинг указателей на интерфейсы мы рассмотрим ниже.

Точки входа DLL должны быть «потокобезопасны»

Компоненту в однопоточном подразделении не нужно быть «потокобезопасным», так как доступ к нему возможен только из создавшего его потока. Однако «потокобезопасны» должны быть точки входа DLL, такие как *DllGetClassObject* и *DllCanUnloadNow*. Функцию *DllGetClassObject* могут одновременно вызывать несколько клиентов из разных потоков. Чтобы сделать эти функции «потокобезопасными», убедитесь, что все совместно используемые данные защищены от параллельного доступа. В некоторых случаях это означает, что фабрика класса также должна быть «потокобезопасна».

Фабрикам класса может понадобиться «потокобезопасность»

Если для каждого компонента Вы создаете отдельную фабрику класса, такой фабрике «потокобезопасность» не требуется, поскольку доступ к ней возможен только для одного клиента. Но если *DllGetClassObject* создает одну фабрику класса, которая используется для порождения всех экземпляров компонента, Вы должны гарантировать

«потокобезопасность» фабрики, поскольку к ней возможен одновременный доступ из разных потоков.

Компонент вне процесса может использовать один экземпляр фабрики класса для создания всех экземпляров компонента. Такая фабрика класса также должна быть «потокобезопасна». Обеспечение потокобезопасности большинства фабрик класса просто, так как они не изменяют никаких совместно используемых данных, кроме счетчика ссылок. Для защиты последних можно использовать *InterlockedIncrement* и *InterlockedDecrement*, что я и демонстрировал уже много лун тому назад в гл. 5.

Удовлетворяющий перечисленным требованиям компонент внутри процесса помечает в Реестре, что поддерживает модель разделенных потоков. О том, как компонент регистрирует свою потоковую модель, рассказывается в конце этой главы, в разделе «Информация о потоковой модели в Реестре». Теперь же мыдетально рассмотрим, что необходимо сделать для маршалинга указателя на интерфейс, который передается другому потоку. Когда компонент в разделенном потоке передает свой интерфейс компоненту в другом потоке, для этогоинтерфейса требуется маршалинг. Неважно, является ли другой поток разделенным или свободным, маршалинг всегда необходим.

Автоматический маршалинг

Во многих случаях COM автоматически выполняет маршалинг интерфейса. В гл. 11 мы рассматривали DLL заместителя/заглушки, которые осуществляют маршалинг интерфейсов между процессами. С точки зрения программиста, потоковая модель не влияет на использование этих DLL. Они автоматически позаботятся о маршалинге между процессами.

DLL заместителя/заглушки используются COM и для маршалинга интерфейсов между разделенным потоком и другими потоками в том же процессе. Таким образом, когда Вы обращаетесь к интерфейсу компонента в другом подразделении, COM автоматически выполняет этот вызов через заместитель, и происходит маршалинг интерфейса.

Ручной маршалинг

Итак, когда же программист должен выполнять маршалинг указателя интерфейса самостоятельно? В основном тогда, когда он пересекает границу подразделения без помощи COM.

Давайте рассмотрим два примера. Сначала пусть клиент создает разделенный поток, создающий компонент и управляющий им. Как в главном потоке, так и в потоке подразделения может потребоваться доступ к такому компоненту. У разделенного потока есть указатель на интерфейс компонента, поскольку этот поток его и создал. Главный поток не может использовать этот указатель напрямую, так как он (поток) находится за пределами подразделения, в котором был создан компонент. Для того, чтобы главный поток мог использовать компонент, разделенный поток должен выполнить маршалинг интерфейса и передать результаты главному потоку. Последний должен выполнить демаршалинг указателя на интерфейс перед использованием. Второй случай имеет место тогда, когда фабрика класса компонента внутри процесса создает его экземпляры в разных потоках. Этот сценарий похож на предыдущий, но теперь создание компонента выполняет в разных потоках сервер (в предыдущем случае это делалклиент). Клиент вызывает *CoCreateInstance*, в результате чего запускается фабрика класса компонента. Когда клиент вызывает *IClassFactory::CreateInstance*, фабрика класса создает новый разделенный поток. Этот новый поток создает компонент. *IClassFactory::CreateInstance* должна вернуть клиенту указатель на интерфейс компонента. Но *CreateInstance* не может

непосредственно передать клиенту указатель на интерфейс, созданный в новом подразделении, так как клиент находится в другом потоке.

Таким образом, поток подразделения должен выполнить маршалинг указателя на интерфейс для *CreateInstance*, которая затем выполняет демаршалинг указателя и возвращает его клиенту.

Самое длинное имя API Win32

Теперь, когда мы узнали, где нужен маршалинг интерфейса, нам нужно знать, как его осуществлять. Вы можете выполнить всю работу сами при помощи функций *CoMarshalInterface* и *CoUnMarshalInterface*. Но если у Вас есть более интересные занятия, используйте вспомогательные функции с самыми длинными именами в API Win32, *CoMarshalInterThreadInterfaceInStream* и *CoGetInterfaceAndReleaseStream*. (Если так пойдет и дальше, скоро имя функции будет занимать целый абзац.)

Использовать эти функции просто. Маршалинг указателя на интерфейс *IX* выполняется так:

```
IStream* pIStream = NULL;
HRESULT hr = CoMarshalInterThreadInterfaceInStream(
            IID_IX, // ID интерфейса, маршалинг которого нужно выполнить
            pIX, // Интерфейс, для которого выполняется маршалинг
            &pIStream); // Поток, куда будут помещены результаты маршалинга
```

Демаршалинг выполняется следующим образом:

```
IX* pIXmarshaled;
HRESULT hr = CoGetInterfaceAndReleaseStream(
            pIStream, // Поток, содержащий интерфейс
            IID_IX, // ID демаршализуемого интерфейса
            (void**)&pIXmarshaled); // Демаршализованный указатель на
      интерфейс
```

Все очень просто, не так ли? Это так просто потому, что COM незаметно для программиста и автоматически использует DLL заместителя/заглушки.

Настало время написать программу

До этого места данная глава носила весьма концептуальный характер, и тому была основательная причина: концепции здесь сложнее реализации. Давайте рассмотрим простой пример. Предположим, Вы хотите в фоновом режиме изменять счетчик в компоненте, и иногда обновлять значение счетчика, выводимое на дисплей. Если бы Вы писали нормальную программу Win32, то создали бы рабочий поток, который бы «фоном» изменял счетчик. Здесь мы будем делать то же самое, но вместо рабочего потока используем разделенный поток. Главный поток создает разделенный поток. Разделенный поток создает компонент и периодически обновляет его счетчик. Этот поток будет передавать главному потоку указатель на интерфейс, чтобы главный поток мог получать и отображать значение счетчика. Все, как в обычном многопоточном программировании в Win32 — за исключением того, что поток подразделения:

- Инициализирует библиотеку COM.
- Имеет собственный цикл выборки сообщений.
- Выполняет маршалинг интерфейса для передачи его обратно главному потоку.

Компонент в точности похож на те, что мы писали ранее.

Теперь самая сложная часть в разработке однопоточного подразделения состоит в том, что у нас есть лишь концепция, а не код. Как обычно, Вы создаете поток. Как обычно, Вы создаете цикл выборки сообщений. Так как я хотел, чтобы подразделение выглядело более «настоящим», то создал для выполнения этих действий небольшой класс *CSimpleApartment*.

CSimpleApartment и *CClientApartment*

CSimpleApartment — это простой класс, инкапсулирующий создание компонента в другом потоке. *CSimpleApartment::StartThread* запускает новый поток.
CSimpleApartment::CreateComponent принимает CLSID компонента и создает его в потоке, запущенном *StartThread*.

Именно здесь все становится интересным (или непонятным). *CSimpleApartment* охватывает оба потока. Часть *CSimpleApartment* вызывается первоначальным потоком, а другая часть — новым потоком. *CsimpleApartment* обеспечивает коммуникацию двух потоков. Поскольку *CSimpleApartment::CreateComponent* вызывается из первоначального потока, постольку она не может создать компонент непосредственно. Компонент надо создать в новом потоке. Поэтому *CreateComponent* использует событие, чтобы дать потоку нового подразделения сигнал к созданию компонента. Для собственно создания поток подразделения вызывает функцию *CreateComponentOnThread*.

CSimpleApartment::CreateComponentOnThread — это чисто виртуальная функция, которую следует определить в производном классе. В этом первом примере производный класс *CclientApartment* реализует версию *CreateComponentOnThread*, которая создает компонент самым обычным способом — при помощи *CoCreateInstance*.

Пример с разделенным потоком

В табл. 13.1 показана структура вызовов функций в коде, который мы собираемся рассматривать. Весь код в правой части таблицы исполняется в разделенномпотоке, созданном *CSimpleApartment::StartThread*.

Таблица 13.1 *Структура вызовов функций в примере с разделенным потоком*

Главный поток	**Разделенный поток**	
WinMain	**CSimpleApartment**	**CSimpleApartment**
InitializeApartment	*StartThread*	*RealThreadProc*
		ClassThreadProc
		CClientApartment::WorkerFunction
	CreateComponent	*CreateComponentOnThread*
		CClientApartment::CreateComponentOnThread

Все самое интересное начинается в CLIENT.CPP с функции *InitializeApartment*. Она вызывает *CSimpleApartment::StartThread*, реализация которой приведена ниже:

```
BOOL CSimpleApartment::StartThread(DWORD WaitTime)
{
  if (IsThreadStarted())
  {
    return FALSE;
  }
```

```
  // Создать поток
  m_hThread = ::CreateThread(NULL, // Защита по умолчанию
                0, // Размер стека по умолчанию
                RealThreadProc,
                (void*)this,
                CREATE_SUSPENDED, // Создать приостановленный
  // поток
                &m_ThreadId); // Получить идентификатор
  // потока
  if (m_hThread == NULL)
  {
    trace("StartThread не может создать поток", GetLastError());
    return FALSE;
  }
  trace("StartThread успешно создала поток");

  // Создать событие для выдачи потоку команды на создание компонента
  m_hCreateComponentEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL);
  if (m_hCreateComponentEvent == NULL)
  {
    return FALSE;
  }

  // Создать событие, которое сигнализируется потоком при завершении
  m_hComponentReadyEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL);
  if (m_hComponentReadyEvent == NULL)
  {
    return FALSE;
  }
  trace("StartThread успешно создала события");

  // Инициализировать время ожидания
  m_WaitTime = WaitTime;
  // Поток был создан приостановленным; запустить его
  DWORD r = ResumeThread(m_hThread);
  assert(r != 0xffffffff);

  // Дождаться начала выполнения потока перед продолжением
  WaitWithMessageLoop(m_hComponentReadyEvent);
  return TRUE;
}
```

CSimpleApartment::StartThread создает новый поток при помощи *::CreateThread*. Она также создает два события для синхронизации двух потоков. Функция *CSimpleApartment::ClassThreadProc*, выполняющаяся в потоке подразделения, использует *m_hComponentReadyEvent* дважды — сначала для сигнализации о том, что новый поток начал выполняться, и в конце для сигнализации о том, что он остановлен. Функция *CSimpleApartment::CreateComponent* использует событие *m_hCreateComponentEvent*, чтобы выдать потоку подразделения команду на вызов *CSimpleApartment::CreateComponentOnThread* для создания компонента. После создания компонента *CreateComponentOnThread* устанавливает *m_hCreateComponentEvent*, чтобы уведомить об окончании создания *CreateComponent*.

CSimpleApartment::WaitWithMessageLoop — это вспомогательная функция, которая ожидает события. Она не просто ждет, а обрабатывает события Windows. Если Вы будете ждать события без обработки сообщений, пользователю покажется, что программа «зависла». Пользовательский интерфейс должен *всегда* обрабатывать сообщения в процесса ожидания. *WaitWithMessageLoop* использует функцию API Win32 *MsgWaitForMultipleObjects*, которую мы рассмотрим ниже.

CSimpleApartment::ClassThreadProc

При запуске потока вызывается статическая функция *RealThreadProc*, которая вызывает *ClassThreadProc*. Windows не может вызывать функции C++, поэтому функции обратного вызова Win32 обязаны быть статическими. При создании потока его процедуре передается указатель нашего класса, чтобы она могла вызвать *ClassThreadProc*. Код *ClassThreadProc* приведен ниже:

```
DWORD CSimpleApartment::ClassThreadProc()
{
  // Инициализировать библиотеку COM
  HRESULT hr = CoInitialize(NULL);
  if (SUCCEEDED(hr))
  {
    // Сигнализировать, что поток запущен
    SetEvent(m_hComponentReadyEvent);
    // Ждать команды на создание компонента
    BOOL bContinue = TRUE;
    while (bContinue )
    {
      switch(::MsgWaitForMultipleObjects(
                  1,
                  &m_hCreateComponentEvent,
                  FALSE,
                  m_WaitTime,
                  QS_ALLINPUT))
      {

        // Создать компонент
        case WAIT_OBJECT_0:
          CreateComponentOnThread();
          break;

        // Обработать сообщения Windows
        case (WAIT_OBJECT_0 + 1):
        MSG msg;
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
          if (msg.message == WM_QUIT)
          {
            bContinue = FALSE;
            break;
          }
          DispatchMessage(&msg);
        }
        break;

        // Выполнить фоновую обработку
        case WAIT_TIMEOUT:
          WorkerFunction();
          break;
        default:
          trace("Ошибка ожидания", GetLastError());
      }
    }
    // Освободить библиотеку COM
    CoUninitialize();
  }
  // Сигнализировать о завершении потока
  SetEvent(m_hComponentReadyEvent);
  return 0;
}
```

Подразделения должны инициализировать библиотеку COM и содержать циклы выборки сообщений. *ClassThreadProc* удовлетворяет этим требованиям. Вместо того, чтобы просто использовать цикл *GetMessage/DispatchMessage*, *ClassThreadProc* использует *MsgWaitForMultipleObjects*, которая ожидает, пока не произойдет одно из трех событий: *m_hCreateComponentEvent*, сообщение Windows или истечение времени тайм-аута. Если устанавливается событие *m_hCreateComponentEvent*, то *MsgWaitForMultipleObjects* прекращает ожидание и *ClassThreadProc* вызывает *CreateComponentOnThread*. Если потоку посылается сообщение Windows, то цикл *PeekMessage/DispatchMessage* извлекает и распределяет это сообщение (а также любые другие, находящиеся в очереди). Если истекает время тайм-аута, вызывается *CSimpleApartment::WorkerFunction*. Эта функция реализована производным классом *CClientApartment*, о котором мы поговорим ниже.

При желании Вы можете использовать *GetMessage/DispatchMessage* в чистом виде. Для выдачи команды на создание компонента вместо события можно использовать *PostThreadMessage*. Однако *MsgWaitForMultipleObjects* более эффективна.

CSimpleApartment::CreateComponent

Теперь, когда мы создали поток, пришла пора создавать компонент. Создание начинается с вызова главным потоком *CSimpleApartment::CreateComponent*. Код этой функции приведен ниже:

```
HRESULT CSimpleApartment::CreateComponent(const CLSID& clsid,
const IID& iid,
IUnknown** ppI)
{
  // Инициализировать совместно используемые данные
  m_pIStream = NULL;
  m_piid = &iid;
  m_pclsid = &clsid;

  // Выдать потоку команду на создание компонента
  SetEvent(m_hCreateComponentEvent);

  // Ожидать завершения создания компонента
  trace("Ожидать завершения создания компонента ");
  if (WaitWithMessageLoop(m_hComponentReadyEvent))
  {
    trace("Ожидание закончилось успешно");
    if (FAILED(m_hr)) // Ошибка GetClassFactory?
    {
      return m_hr;
    }
    if (m_pIStream == NULL) // Ошибка при маршалинге?
    {
      return E_FAIL;
    }

    trace("Демаршалинг указателя на интерфейс");
    // Выполнить демаршалинг интерфейса
    HRESULT hr = ::CoGetInterfaceAndReleaseStream(m_pIStream,
                        iid,
                        (void**)ppI);
    m_pIStream = NULL;
    if (FAILED(hr))
    {
      trace("Ошибка CoGetInterfaceAndReleaseStream", hr);
      return E_FAIL;
    }
```

```
    return S_OK;
  }
  trace("Что случилось?");
  return E_FAIL;
}
```

Функция *CreateComponent* выполняет четыре основных действия. Во-первых, она копирует свои параметры в переменные-члены. Во-вторых, она выдает потоку команду на создание компонента. В-третьих, она ждет завершения создания компонента. И в-четвертых, она выполняет демаршалинг запрошенного интерфейса компонента.

CSimpleApartment::CreateComponentOnThread

Когда *CreateComponent* устанавливает *m_hCreateComponentEvent*, *ClassThreadProc* вызывает приватную внутреннюю версию *CreateComponentOnThread*, а та выполняет два основных действия. Во-первых, она вызывает чисто виртуальную версию *CreateComponentOnThread* с параметрами, которые были переданы *CreateComponent*. Непосредственная передача параметров *CreateComponentOnThread* упрощает ее реализацию в производном классе. Во-вторых, она выполняет маршалинг интерфейса:

```
void CSimpleApartment::CreateComponentOnThread()
{
  IUnknown* pI = NULL;
  // Вызвать производный класс для фактического создания компонента
  m_hr = CreateComponentOnThread(*m_pclsid, *m_piid, &pI);
  if (SUCCEEDED(m_hr))
  {
    trace("Компонент создан успешно");
    // Выполнить маршалинг интерфейса для основного потока
    HRESULT hr = ::CoMarshalInterThreadInterfaceInStream(*m_piid,
                                         pI,
                                         &m_pIStream);
     assert(SUCCEEDED(hr));

     // Освободить указатель pI
     pI->Release();
   }
   else
   {
   trace("Ошибка CreateComponentOnThread", m_hr);
  }
  trace("Сигнализировать главному потоку, что компонент создан");
  SetEvent(m_hComponentReadyEvent);
}
```

CreateComponentOnThread использует функцию *CoMarshalInterThreadInterfaceInStream* для маршалинга указателя на интерфейс в другой поток. Код *CreateComponent* выполняет демаршалинг интерфейса.

CClientApartment

В этом примере *CClientApartment* реализует две виртуальные функции: *CreateComponentOnThread* и *WorkerFunction*. *CClientApartment* предназначена для использования клиентами, которые хотят создавать компоненты в разных потоках. Она переопределяет *CreateComponentOnThread*, чтобы вызвать *CoCreateInstance*:

```
HRESULT CClientApartment::CreateComponentOnThread(const CLSID& clsid,
const IID& iid,
IUnknown** ppI)
{
  HRESULT hr = ::CoCreateInstance(clsid,
                        NULL,
                        CLSCTX_INPROC_SERVER,
                        iid,
                        (void**)ppI);
  if (SUCCEEDED(hr))
  {
    // Запросить интерфейс IX, который используется в WorkerFunction
    hr = (*ppI)->QueryInterface(IID_IX, (void**)&m_pIX);
    if (FAILED(hr))
    {
      // Если мы не можем с этим работать, на дадим и другим
      (*ppI)->Release();
      return E_FAIL;
    }
  }
  return hr;
}
```

CClientApartment::CreateComponentOnThread запрашивает у созданного ею компонента интерфейс *IX*, который используется затем функцией *WorkerFunction*:

```
void CClientApartment::WorkerFunction()
{
  if (m_pIX)
  {
    m_pIX->Tick();
  }
}
```

CLIENT.CPP

Теперь созданы и поток и компонент. Всякий раз, когда истекает интервал времени *CSimpleApartment::m_WaitTime*, *CSimpleApartment::ClassThreadProc* вызывает *CClientApartment::WorkerFunction*. Таким образом, наш компонент обновляется каждые несколько миллисекунд. Для отображения этих изменений в своем окне клиент создает таймер. Получив сообщение WM_TIMER, клиент вызывает *OnTick*, которая обращается к *IX::GetCurrentCount* и затем отображает значение счетчика в окне. Когда клиент вызывает *IX::GetCurrentCount*, происходит маршалинг вызова через границу подразделения. Когда же *WorkerFunction* вызывает *IX::Tick*, имеет место вызов из того же самого подразделения, и маршалинг не производится. Разделенные потоки могут создаваться не только клиентами. Можно разработать компоненты для создания разделенных потоков.

Фактически можно создать фабрику класса, которая создает компоненты в разных разделенных потоках. Вот и все. Как видите, самая сложная часть реализации разделенного потока — это создание и управление потоками.

Теперь, когда мы стали экспертами по разделенным потокам, давайте рассмотрим модель свободных потоков.

Реализация модели свободных потоков

Если Вам приходилось писать многопоточные программы, то свободные потоки вряд ли составят для Вас по-настоящему новые проблемы. Свободные потоки создаются и управляются обычными функциями Win32 для работы с потоками, такими как *CreateThread*, *ResumeThread*, *WaitForMultipleObjects*, *WaitForSingleObject*, *CreateMutex* и

CreateEvent. Используя стандартные синхронизационные объекты — мьютексы, критические секции и семафоры, — Вы можете управлять доступом к внутренним данным своего компонента, сделав его «потокобезопасным». Хотя обеспечить настоящую потокобезопасность компонента — всегда непростая задача, хорошо разработанный интерфейс COM совершенно ясно покажет Вам, когда происходит доступ к компоненту.

Если Вы еще не писали многопоточных программ, то пример из этого раздела — хорошая отправная точка для обучения. Мы будем использовать несколько мютексов, чтобы предотвратить одновременный доступ нескольких потоков к один и тем же данным. Помимо «потокобезопасности» компонентов, для использования свободных потоков надо выполнить, по существу, только три требования. Первое состоит в том, что Ваша операционная система должна поддерживать модель свободных потоков COM. Ее поддерживает Windows NT 4.0 и Windows 95 тоже, если Вы установили расширения DCOM. В гл. 11 мы рассматривали, как программным путем определить, что операционная система поддерживает свободные потоки. (В основном для этого нужно установить наличие в OLE32.DLL функции *CoInitializeEx*.)

Говоря о *CoInitializeEx*: поток должен вызвать эту функцию с параметром COINIT_MULTITHREADED, чтобы обозначить себя как свободный. Что значит объявить поток свободным? Поток, создающий компонент, определяет, как компонент обрабатывает вызовы из других потоков. Если компонент создается свободным
потоком, то он может быть вызван любым другим свободным потоком в любой момент.
После того, как поток вызывал *CoInitializeEx* с параметром COINIT_MULTITHREADED, он не может вызвать ее с другим параметром. Поскольку *OleInitialize* вызывает *CoInitializeEx* с параметром COINIT_APARTMENTTHREADED, постольку Вы не можете использовать библиотеку OLE из свободного потока.

Третье требование является фактически требованием не свободных, а разделенных потоков. Необходимо выполнять маршалинг указателей на интерфейсы при передаче их разделенным потокам. Кстати, это имеет значение только в том случае, если указатель передается не посредством интерфейса COM. Если указатель передается через интерфейс COM, то COM выполняет маршалинг автоматически. Если клиент находится в другом процессе, то и здесь COM выполняет маршалинг автоматически. Конечно, для автоматического маршалинга Вы должны предоставить COM DLL заместителя/заглушки. Маршалинг между разделенными потоками мы обсуждали в предыдущем разделе. Свободные потоки используют для маршалинга интерфейсов вручную те же самые функции *CoMarshalInterThreadInterfaceInStream* и *CoGetInterfaceAndReleaseStream*. Как мы увидим далее, COM может незаметно для нас оптимизировать маршалинг.
Для компонентов внутри процесса имеется четвертое требование. Они должны регистрировать себя в Реестре в качестве поддерживающих свободные потоки. Мы рассмотрим этот пункт в разделе «Информация о потоковой модели в Реестре».

Как видите, за исключением маршалинга интерфейсов в другие подразделения, требования модели свободных потоков просты. Самая сложная задача, связанная с этой моделью, состоит в обеспечении «потокобезопасности» компонентов. Однако это не требование COM, а стандартная проблема многопоточности.

Пример со свободным потоком

Создание свободного потока не слишком отличается от создания разделенного потока. Первый свободный поток увеличивает счетчик компонента (как в примере с разделенным потоком). Другой счетчик уменьшает его. Кроме того, теперь мы будем представлять себе счетчик как находящийся то на одной, то на другой «стороне». Первый свободный поток переводит его «налево», а второй — «направо».

Главный поток (разделенный) получает после маршалинга копию указателя на интерфейс и использует ее для периодического опроса состояния компонента. Большая

часть кода аналогична приведенномуранее для создания разделенного потока. Чтобы не повторяться, я лишь укажу на отличия в этих двух примерах.

Очевидные отличия

Самое очевидное отличие — замена имени *CSimpleApartment* на *CSimpleFree*. При создании свободного потока создается не новое подразделение, а лишь поток. Аналогично, *CClientApartment* теперь называется *CClientFree*. Подчеркну, что *CSimpleFree* — не универсальный подход к созданию и управлению свободными потоками. Сам по себе *CSimpleFree* не «потокобезопасен». Он предназначен только для создания свободных потоков клиентом, использующим модель разделенных потоков. Недостаток устойчивости *CSimpleFree* компенсируется простотой.

CSimpleFree::ClassThreadProc

Единственная функция, которой *CSimpleFree* существенно отличается от *CSimpleApartment*, — это *ClassThreadProc*. Вместо вызова *CoInitialize*, как это делается в *CSimpleApartment*, она вызывает *CoInitializeEx(0, COINIT_MULTITHREADED)*. Прежде чем использовать *CoInitializeEx*, необходимо сделать две вещи. Во-первых, нужно определить *_WIN32_WINNT = 0x400* или *_WIN32_DCOM*. Если этого не сделать, то в OBJBASE.H не будет определения *CoInitializeEx*. Во-вторых, мы должны во время выполнения программы убедиться, что операционная система поддерживает *CoInitializeEx*. Все это показано ниже:

```
BOOL CSimpleFree::ClassThreadProc()
{
  BOOL bReturn = FALSE;

  // Проверить наличие CoInitializeEx
  typedef HRESULT (__stdcall *FPCOMINITIALIZE)(void*, DWORD);
  FPCOMINITIALIZE pCoInitializeEx =
  reinterpret_cast<FPCOMINITIALIZE>(
  ::GetProcAddress(::GetModuleHandle("ole32"), "CoInitializeEx"));
  if (pCoInitializeEx == NULL)
  {
    trace("Эта программа требует поддержки свободных потоков в DCOM");
    SetEvent(m_hComponentReadyEvent);
    return FALSE;
  }

  // Инициализировать библиотеку COM
  HRESULT hr = pCoInitializeEx(0, COINIT_MULTITHREADED);
  if (SUCCEEDED(hr))
  {
    // Сигнал о начале работы
    SetEvent(m_hComponentReadyEvent);

    // Создать массив событий
    HANDLE hEventArray[2] = { m_hCreateComponentEvent,
    m_hStopThreadEvent };

    // Ждать команды на создание компонента
    BOOL bContinue = TRUE;
    while (bContinue)
    {
      switch(::WaitForMultipleObjects(2,
                 hEventArray,
                 FALSE,
                 m_WaitTime))
```

```
    {
      // Создать компонент
      case WAIT_OBJECT_0:
        CreateComponentOnThread();
        break;

        // Остановить поток
        case (WAIT_OBJECT_0 +1):
          bContinue = FALSE;
          bReturn = TRUE;
          break;

        // Выполнить фоновую обработку
        case WAIT_TIMEOUT:
          WorkerFunction();
          break;
        default:
          trace("Ошибка при ожидании", GetLastError());
      }
    }
    // Освободить библиотеку COM
    CoUninitialize();
  }
  // Сигнализировать, что мы закончили
  SetEvent(m_hComponentReadyEvent);
  return bReturn;
}
```

Поскольку *CSimpleFree* создает свободные потоки, ей не нужен цикл выборки сообщений. Поэтому я заменил *MsgWaitForMultipleObjects* на *WaitForMultipleObjects*. Для остановки потока вместо WM_QUIT используется *m_hStopThreadEvent*.

Хотя *MsgWaitForMultipleObjects* больше не нужна нам в *ClassThreadProc*, она по-прежнему используется в *CSimpleFree::StartThread* и в *CSimpleFree::CreateComponent*. Эти функции вызываются главным потоком (разделенным), поэтому они должны обрабатывать сообщения, чтобы не блокировать пользовательский интерфейс.
Фактически только в этом и состоят различия между *CSimpleFree* и *CSimpleApartment*.

CClientFree

Я хочу продемонстрировать Вам работу двух свободных потоков, которые совместно используют один компонент, без маршалинга их указателей на интерфейсы. Для этого я добавил в *CClientFree* два метода. *CClientFree* в этом примере со свободным потоком служит эквивалентом *CClientApartment* из предыдущего примера. *CClientFree* наследует *CSimpleFree* и реализует виртуальные функции *CreateComponentOnThread* и *WorkerFunction*. В *CClientFree* две новые функции — *ShareUnmarshaledInterfacePointer* и *UseUnmarshaledInterfacePointer*. (Меня до того воодушевили длинные имена некоторых функций COM, что я решил так называть и свои функции.) Первая, *ShareUnmarshaledInterfacePointer*, возвращает указатель на интерфейс *IX*, используемый *CClientFree* в его функции *WorkerFunction*. Маршалинг этого интерфейса не выполняется, поэтому его можно использовать только в свободном потоке. Вторая функция, *UseUnmarshaledInterfacePointer*, устанавливает указатель на *IX*, который объект *CClientFree* будет использовать в своей функции *WorkerFunction*. Теперь посмотрим, как эти функции используются в CLIENT.CPP.

Функция *InitializeThread* используется в CLIENT.CPP для создания свободного потока и компонента. Эта функция похожа на вызов *InitializeApartment* из примера однопоточного подразделения. После вызова *InitializeThread* клиент вызывает *InitializeThread2*. Эта функция создает второй поток. Однако вместо создания

второго компонента этот поток использует компонент, созданный первым потоком. Код *InitializeThread2* показан ниже:

```
BOOL InitializeThread2()
{
  if (g_pThread == NULL)
  {
    return FALSE;
  }

  // Создать второй поток
  // У этого потока другая WorkerFunction
  g_pThread2 = new CClientFree2;

  // Запустить поток
  if (g_pThread2->StartThread())
  {
    trace("Второй поток получен успешно");
    // Получить тот же указатель, который использует первый поток
    IX* pIX = NULL;
    pIX = g_pThread->ShareUnmarshaledInterfacePointer();
    assert(pIX != NULL);

    // Использовать этот указатель во втором потоке
    g_pThread2->UseUnmarshaledInterfacePointer(pIX);
    pIX->Release();
    return TRUE;
  }
  else
  {
    trace("Ошибка при запуске второго потока");
    return FALSE;
  }
}
```

InitializeThread2 вместо объекта *CClientFree* создает объект *CClientFree2*. *CClientFree2* отличается от *CclientFree* только реализацией *WorkerFunction*. Обе реализации приведены ниже:

```
void CClientFree::WorkerFunction()
{
  CSimpleLock Lock(m_hInterfaceMutex);
  if (m_pIX)
  {
    m_pIX->Tick(1);
    m_pIX->Left();
  }
}

void CClientFree2::WorkerFunction()
{
  CSimpleLock Lock(m_hInterfaceMutex);
  if (m_pIX)
  {
    m_pIX->Tick(-1);
    m_pIX->Right();
  }
}
```

CSimpleLock мы скоро обсудим. Я изменил *IX::Tick*, чтобы она принимала в качестве параметра размер увеличения счетчика. Я также добавил методы *Left* и *Right*. Эти функции управляют тем, на какой «стороне» находится счетчик. *CClientFree* увеличивает

счетчик и помещает его «налево». *CClientFree2* уменьшает его и помещает «направо». Функция *InRightHand* возвращает TRUE, если счетчик находится на правой стороне. Таким образом, с ее помощью мы можем определить, какой поток использовал компонент последним.

Изменения в компоненте

Помимо добавления к компоненту нескольких методов мы также должны сделать его «потокобезопасным». В конце концов, у нас два разных потока одновременно увеличивают и уменьшают один счетчик. Для того, чтобы обеспечить защиту компонента, я ввел простой класс *CsimpleLock*:

```
class CSimpleLock
{
public:
  // Заблокировать
  CSimpleLock(HANDLE hMutex)
  {
    m_hMutex = hMutex;
    WaitForSingleObject(hMutex, INFINITE);
  }
  // Разблокировать
  ~CSimpleLock()
  {
    ReleaseMutex(m_hMutex);
  }
private:
  HANDLE m_hMutex;
};
```

Конструктору *CSimpleLock* передается описатель мьютекса. Конструктор не возвращает управление, пока не дождется мьютекса. Деструктор *CsimpleLock* освобождает мьютекс, когда поток управления выходит из области действия переменной. Для защиты функции нужно просто создать объект *CSimpleLock*:

```
HRESULT __stdcall CA::Tick(int delta)
{
  CSimpleLock Lock(m_hCountMutex);
  m_count += delta;
  return S_OK;
}

HRESULT __stdcall CA::Left()
{
  CSimpleLock Lock(m_hHandMutex);
  m_bRightHand = FALSE;
  return S_OK;
}
```

Наш компонент использует два разных мьютекса — *m_hHandMutex* и *m_hCountMutex*. Один из них защищает счетчик, а второй — переменную, указывающую сторону. Наличие двух разных мьютексов позволяет одному потоку работать с переменной, указывающей сторону, пока второй работает со счетчиком. Доступ к компонентам в подразделении возможен только для одного потока — потока этого подразделения. Если бы компонент выполнялся в потоке подразделения, один поток не смог бы вызвать *Left*, если другой уже вызывает *Tick*. Однако при использовании свободных потоков синхронизация возлагается на разработчика компонента, который

может использовать свое знание внутреннего устройства компонента для оптимальной синхронизации.

Оптимизация маршалинга для свободных потоков

Как маршалинг, так и синхронизация работают медленно. Если возможно, избегайте их. Одно из правил, связанных с разделенными потоками, — необходимость маршалинга интерфейсов перед передачей разделенным потокам. Но предположим, что клиент в разделенном потоке хочет использовать интерфейс компонента свободных потоков в том же самом процессе. Нам в действительности не нужен маршалинг, так как процесс один и тот же. Нам также не нужна синхронизация вызовов нашего компонента, выполняемая COM; в конце концов, мы сделали компонент «потокобезопасным», чтобы его можно было использовать из нескольких потоков одновременно. Похоже, компоненты в свободном потоке должны уметь напрямую передавать указатели на интерфейсы другим разделенным потокам в том же самом процессе. Да, они это умеют.

Оптимизация не просто возможна — библиотека COM ещеи предоставляет специальный агрегируемый компонент, который выполнит для Вас эту оптимизацию. *CoCreateFreeThreadedMarshaler* создает компонент с интерфейсом *IMarshal*, который определяет, находится ли клиент интерфейса в том же самом процессе. Если это так, то при маршалинге указатели передаются без изменений. Если клиент находится в другом процессе, то выполняется стандартный маршалинг интерфейса. Самое замечательное в *CoCreateFreeThreadedMarshaler* то, что Вам не нужно знать, кто является клиентом, — все чудеса происходят автоматически. Эта оптимизация работает и в сочетании с *CoMarshalInterThreadInterfaceInStream* и *CoGetInterfaceAndReleaseStream*. Это позволяет Вам выполнять явный маршалинг своих интерфейсов и предоставить заботу об оптимизации COM.

Ниже приведен код, создающий маршалер свободных потоков. Также показана реализация *QueryInterface*, которая делегирует запросы на *IMarshal* маршалеру свободных потоков.

```
HRESULT CA::Init()
{
  HRESULT hr = CUnknown::Init();
  if (FAILED(hr))
  {
    return hr;
  }

  // Создать мьютекс для защиты счетчика
  m_hCountMutex = CreateMutex(0, FALSE, 0);
  if (m_hCountMutex == NULL)
  {
    return E_FAIL;
  }

  // Создать мьютекс для защиты индикатора стороны
  m_hHandMutex = CreateMutex(0, FALSE, 0);
  if (m_hHandMutex == NULL)
  {
    return E_FAIL;
  }

  // Агрегировать маршален свободных потоков
  hr = ::CoCreateFreeThreadedMarshaler(
                GetOuterUnknown(),
                &m_pIUnknownFreeThreadedMarshaler);
  if (FAILED(hr))
```

```
  {
    return E_FAIL;
  }
  return S_OK;
}

HRESULT __stdcall CA::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IX)
  {
    return FinishQI(static_cast<IX*>(this), ppv);
  }
  else if (iid == IID_IMarshal)
  {
    return m_pIUnknownFreeThreadedMarshaler->QueryInterface(iid, ppv);
  }
  else
  {
    return CUnknown::NondelegatingQueryInterface(iid, ppv);
  }
}
```

Замечание о терминологии

Как я говорил в начале главы, терминология COM, связанная с потоками, существенно различается в разных документах. Авторы Win32 SDK используют слово «apartment» («подразделение») не совсем так, как это делаю я. То, что я называю подразделением, они называют «многопоточное подразделение» для обозначения всей совокупности свободных потоков. В их терминах, в процессе может быть произвольное число «однопоточных подразделений», но только одно «многопоточное подразделение». Я надеюсь, что это разъяснение поможет Вам избежать путаницы при чтении документации Win32 SDK.

Информация о потоковой модели в Реестре

COM необходимо знать, какую потоковую модель поддерживают компоненты внутри процесса, чтобы обеспечить правильный маршалинг их интерфейсов и синхронизацию при вызовах между потоками. Чтобы зарегистрировать потоковую модель своего компонента внутри процесса, добавьте в раздел компонента *InprocServer32* параметр с именем *ThreadingModel*. (*ThreadingModel* — это именованный параметр, а не подраздел!) Для *ThreadingModel* допускается одно из трех значений: *Apartment*, *Free* или *Both*.

Должно быть очевидно, что компоненты, которые можно использовать в разделенных потоках, устанавливают этот параметр в значение *Apartment*. Компоненты, которые можно использовать в свободных потоках, задают значение *Free*. Компоненты, которые могут использоваться как разделенными, так и свободными потоками, используют значение *Both*. Если компонент ничего не знает о потоках, то параметр не задан вообще. Если параметр не существует, то подразумевается, что компонент не поддерживает многопоточность. Все компоненты, обслуживаемые данным сервером внутри процесса, должны иметь одну и ту же потоковую модель.

Резюме

В одной главе мы не только научились реализовывать разделенные и свободные потоки, но также узнали, что такое подразделение. Подразделение — это концептуальный конгломерат, состоящий из потока и цикла выборки сообщений. Поток подразделения

похож на типичный процесс Win32 в том смысле, что оба имеют один поток и цикл выборки сообщений. В одном процессе может быть любое число подразделений и свободных потоков.

Разделенные потоки должны инициализировать COM, иметь цикл выборки сообщений и выполнять маршалинг указателей на интерфейсы в другие потоки. Компонент, созданный в разделенном потоке, должен вызываться только создавшим его потоком. Аналогичное правило существует и для оконной процедуры. Серверы внутри процесса должны иметь «потокобезопасные» точки входа, но компоненты могут не быть «потокобезопасными», так как синхронизацию обеспечивает COM.

Свободные потоки должны инициализировать COM при помощи *CoInitializeEx*. Они не обязаны иметь цикл выборки сообщений, но по-прежнему обязаны выполнять маршалинг интерфейсов в разделенные потоки и в другие процессы. Им не нужно выполнять маршалинг интерфейсов в другие свободные потоки в том же самом процессе.

Встает вопрос, потоки какого типа следует использовать Вам? Код пользовательского интерфейса обязан использовать разделенные потоки. Это гарантирует, что сообщения будут обработаны и у пользователя не возникнет впечатления, что программа «зависла». Если Вы хотите выполнять в фоновом режиме только простейшие операции, следует использовать разделенные потоки. Их гораздо легче реализовывать, так как не нужно заботиться о «потокобезопасности» используемых в них компонентах.

Однако в любом случае для всех вызовов разделенного потока необходим маршалинг. Это может существенно снизить производительность. Поэтому, если Вам необходим интенсивный обмен информацией между разными потоками, либо соберите весь код в один поток, либо используйте свободные потоки. Вызовы между свободными потоками внутри одного процесса не требуют маршалинга и могут выполняться значительно быстрее, в зависимости от того, как реализована синхронизация внутри компонента.

14. Заключение

Основа COM — стандартные интерфейсы

Как я уже неоднократно повторял, COM основана на интерфейсах. Чем больше компонентов используют одни и те же интерфейсы, тем больше вероятность полиморфного использования компонентов. Многие интерфейсы уже определены COM, OLE, управляющими элементами ActiveX, документами ActiveX и Автоматизацией.
Разработчику компонента COM следует изучить эти уже существующие интерфейсы. Даже если Вы решите не использовать их в своем приложении, то многое узнаете о создании с их помощью гибких компонентных архитектур.

Итак, мы подошли к концу. Вы знаете, как создавать интерфейсы COM на C++, реализовывать *IUnknown* и *IClassFactory* и регистрировать свои компоненты в Реестре Windows. Вам также известно, как создавать приложения из компонентов, включающих и агрегирующих другие компоненты. Вы знаете, как упростить себе жизнь с помощью классов C++ и smart-указателей. Вы также умеете описывать свои интерфейсы в файлах IDL, чтобы автоматически генерировать библиотеки маршалинга и библиотеки типа. Реализация *IDispatch* — это простой процесс, состоящий в использовании *ITypeInfo*. Наконец, Вы мастерски умеете создавать компоненты, реализующие модель разделенных потоков.

Если Вы решите написать компонент COM, то, учитывая Ваши знания, единственным недостающим ингредиентом будут конкретные интерфейсы COM. Вы знаете, как реализовать интерфейс. Теперь Вам нужно либо разработать собственный интерфейс, либо найти стандартный и реализовать его. Microsoft уже разработаны
сотни интерфейсов для технологий ActiveX, DirectX и OLE. Управляющий элемент ActiveX — это просто реализация набора интерфейсов. Документ ActiveX — также набор интерфейсов с их реализациями. Управляющие элементы и документы ActiveX используют много общих стандартных интерфейсов. Реализация интерфейсов ActiveX, DirectX и OLE — непростая задача. Однако все это, как говорится, детали реализации. Проблема заключается не в COM, ведь после этой книги Вы стали настоящим экспертом по COM.

15. Список литературы

1. Архитектура Windows для разработчиков /Microsoft press. - М.: 1998.
2. Основы COM /Дейл Роджерсон, Microsoft Press. – М.: 1999.
3. The Component Object Model Specification /Microsoft.: 1995.