Дональд Бокс
Сущность технологии COM. Библиотека программиста

Дональд Бокс
Сущность технологии COM. Библиотека программиста

Посвящается Юдит С., которая помогла мне одолеть одну вещь, более устрашающую, чем COM, и сделала возможной написание этой книги, и Барбаре, которая оставалась со мной достаточно долго для того, чтобы увидеть, чем все это кончилось.

Предисловие Чарли Киндела

Когда я сел писать это предисловие, мне не давали покоя следующие мысли:

Будет ли портрет Дона на четвертой стороне обложки, и если да, то какой длины будут его волосы?

Осознают ли читатели этой книги, что у Дона есть индивидуализированные (personalized) лицензионные платы, способные читать интерфейс «IUNKNOWN»?

Что за чертовщину нужно писать в предисловии к книге?

У меня было две идеи насчет того, что написать в этом предисловии. Первая – высказать несколько мыслей о конструировании COM, которые я уже давно собираюсь записать. Вторая идея – польстить Дону Боксу в той же мере, в какой он польстил мне обращением с просьбой написать предисловие к своей книге. В конце концов я решил осуществить обе идеи.

Что есть COM? Зачем его придумали? Дон кратко осветил эти вопросы в первой главе. Вводная часть заканчивается словами «…в этой главе показана архитектура для повторного использования модулей, которая позволяет динамично и эффективно строить системы из независимо сконструированных двоичных компонентов». Остальная часть этой главы ведет вас шаг за шагом сквозь мыслительный процесс, происходивший в умах разработчиков COM с 1988 по 1993 годы, когда была выпущена первая версия COM.

Я думаю, что существует несколько аспектов конструирования COM, которые

обеспечили его длительный успех. Первое и основное – это практичность, второе – простота, из которой проистекает его гибкость, или податливость.

Практичность

COM относится к разработке программного обеспечения весьма прагматично. Вместо того чтобы искать решение на основе почти религиозной академической догмы объектно-ориентированного программирования, COM-конструирование принимает во внимание как человеческую природу, так и капитализм. Команда разработчиков выделила лучшие, наиболее *коммерчески* убедительные аспекты классического объектного ориентирования (ОО) и объединила их с тем, чему она научилась при попытках добиться повторного использования предыдущих программных разработок – как внутри, так и вне Microsoft.

Большинство классических текстов, посвященных ОО, описывают систему или язык как ориентированный объект, если он поддерживает инкапсуляцию (сокрытие информации), полиморфизм и наследование. Часто подчеркивается, что главной движущей силой повторного использования является наследование. Разработчики COM не согласились с таким акцентом. Они поняли, что это слишком упрощенное представление и что в действительности существуют два вида наследования. Наследование реализации предполагает, что наследуется фактическая реализация (поведение). Наследование интерфейсов предполагает, что наследуется только определение (спецификация) поведения. Именно второй вид наследования обеспечивает полиморфизм, и этот вид полностью поддерживается моделью COM. С другой стороны, наследование реализации – это просто один из механизмов для повторного использования существующей реализации. Тем не менее, если конечной целью является повторное использование, тогда наследование реализации является просто средством для достижения этой цели, но не является самоцелью.

Как в исследовательских, так и в коммерческих кругах разработчиков программного обеспечения считалось общепринятым, что наследование реализации – полезный и мощный инструмент, хотя он может привести к чрезмерной связи между базовым и производным классами. Поскольку наследование реализации часто вызывает «утечку» некоторых элементов реализации базового класса, нарушая инкапсуляцию этого класса, разработчики COM понимали, что наследование реализации должно быть ограничено программированием *внутри* компонентов. Поэтому COM не поддерживает наследование реализации *между* компонентами, но поддерживает ее внутри компонентов. Наследование же интерфейсов COM поддерживает полностью (по сути, она полагается на это).

Разработчики COM развенчали миф о том, что главную роль при достижении повторного использования играет наследование. Фундаментальное понятие, использующееся в COM при моделировании повторного использования, – это инкапсуляция, а не наследование. А принцип наследования COM использует при моделировании взаимоотношения типов между объектами, выполняющими сходные функции. Построением COM-модели повторного использования на основе инкапсуляции разработчики поддерживали повторное использование в форме *черного ящика* , устраивающее ожидаемый рынок компонентов. Идея состоит в том, что клиенты должны иметь дело с объектами как с непрозрачными компонентами в смысле того, что находится у них внутри и как они реализованы. Разработчики COM полагали, что для проведения этой идеи в жизнь должна быть разработана архитектура. С какой стати любой может разрабатывать систему с другой моделью для повторного использования? Хороший вопрос. Дело, однако, в том, что мир полон «объектно-ориентированных» систем, которые не только не поддерживают инкапсуляцию в стиле черного ящика, но даже затрудняют ее достижение. Классическим примером этого является C++. В первой главе своей книги Дон очень понятно объясняет то, что я подразумеваю под этим.

Следующие уравнения иллюстрируют различия между объектно-ориентированным и

компонентно-ориентированным программированием.

Объектно-ориентированное программирование = полиморфизм + (немного) позднее связывание + (немного) инкапсуляции + наследование.

Компонентно-ориентированное программирование = полиморфизм + (истинно) позднее связывание + (действительная, принудительная) инкапсуляция + наследование интерфейсов + двоичное повторное использование.

Во всяком случае для меня эта дискуссия – род забавы. Борцы за чистоту ОО, проживающие в comp.object и comp.object.corba, выбились из сил, тыча пальцами в COM и говоря: «Но он не *по-настоящему* объектно-ориентированный». Вы можете оспорить это двумя способами:

1. «Он-то как раз по-настоящему! Это *ваше* определение ОО неправильное».

2. *«Ну и что!?!* COM имеет феноменальный коммерческий успех и позволяет тысячам независимых разработчиков создавать потрясающее программное обеспечение, которое интерполирует и интегрирует. И они делают деньги. Много денег[1]. Программные компоненты, написанные ими, покупаются, используются и повторно используются. Разве не в *этом* смысл любой технологии? Кроме того, я всегда могу доказать, что *только* COM является истинно компонентно-ориентированным [2].

Вот так-то!

Простота ведет к податливости (malleability)

mal-le-a-ble (mal'e-e-bel) *adjective (прилагательное)*

1. Способный быть выкованным или сформированным, как под ударами молота или под давлением: *податливый металл.*

2. Легко контролируемый или поддающийся влиянию; послушный.

3. Способный подстраиваться под изменяющиеся обстоятельства, легко приспосабливаемый: *гибкий ум прагматика* [3].

Первое реальное применение COM заключалось в том, что он был взят за основу при второй попытке фирмы Microsoft создать сложную структуру документов Object Linking & Embedding 2.0 (связывание и внедрение объектов, OLE 2.0). Если вы рассмотрите всю массу возможностей для приложения COM в настоящее время, то сразу поймете, что я имею в виду, называя его податливым. Программисты используют COM для обеспечения сменной (plug-in) архитектуры для своих приложений; для конструирования крупномасштабных многоярусных клиент/серверных приложений; для ведения дел и заключения сделок в мире бизнеса; для создания развлекательных сюжетов на Web-страницах; для контроля и мониторинга производственных процессов и даже для выслеживания спутников-шпионов путем дистанционного управления целой армией телескопов.

---

[1] *1* Исследование Гиги (Giga), порученное ему фирмой Microsoft, показывает, что в 1997 году рынок коммерческих компонентов, основанных на COM, составил 410 миллионов долларов. Ожидается, что этот рынок к 2001 году превысит 2.8 миллиарда долларов. Эти цифры не включают продукцию Microsoft.

[2] *2* Упражнение для читателя: назовите одну коммерчески доступную объектную систему, кроме COM, которая предусматривает двоичное повторное использование, поддерживает жесткое управление версиями, прозрачность адресации и независимость от языка программирования. Если вы скажете: «CORBA», – то вас надули и вы не знаете такой объектной системы.

[3] *3* Выдержки из *The American Heritage Dictionary of the English Language, Third Edition* © 1996 by Houghton Mifflin Company. Электронная версия лицензирована INSO Corporation; дальнейшее копирование и распространение осуществляется в соответствии с законом об авторских правах Соединенных Штатов. Все права защищены.

Эта податливость достигнута благодаря тому, что разработчики COM поставили во главу угла принцип: ядро модели будет настолько простым, насколько это необходимо, но не более. Одной из наиболее явных сторон этого подхода является нудность программирования на COM на сегодняшний день. Программисты, работающие на C или C++, должны возиться со всей этой ерундой, в том числе с GUID и со счетчиками ссылок. Можно было бы добавить к COM всякого рода усовершенствования с целью упрощения работы с ней. Но разработчики вместо этого акцентировали внимание на том, чтобы заставить модель работать. Они считали, что если они достигнут успеха, то сервисную поддержку можно будет обеспечить позднее. И это предположение было подтверждено недавними выпусками простых в употреблении инструментов COM, таких как Visual Basic, поддержка COM в Visual C++ 5.0, а также Active Template Library. К тому времени, когда вы читаете этот текст, фирма Microsoft должна уже объявить о своих будущих планах радикально упростить разработку COM с помощью внедрения общего времени выполнения (runtime), которое будет доступно для всего инструментария: COM+.

Фольклор

Любая технология, распространенная так широко, как COM, начинает обрастать фольклором. Ради забавы приведу несколько, возможно, неизвестных вам тезисов. Некоторые из них даже правдивы.

Огромное множество людей из различных групп по всей фирме Microsoft внесли серьезный вклад в разработку COM, но главными архитекторами COM были Боб Аткинсон (Bob Atkinson), Тони Вильямс (Tony Williams) и Крейг Виттенберг (Craig Wittenberg). Все трое по-прежнему в Microsoft за работой над воистину редкостной чепухой.

Боб, Тони и Крейг были частью кросс-группы, получившей привилегию создания базовой технологии, которая позволила бы воплотить в жизнь мечту Билла Гейтса о IAYF (Information at Your Fingertips – информация на кончиках ваших пальцев)[4]. Но хотя эти трое прекрасно осознавали грядущую мощь COM, на деле они были обременены выпуском того, что лишь использовало COM: OLE 2.0. Это помогает объяснить, почему оформление документации для собственно COM заняло столько времени. Жаль.

Первая реализация COM была выпущена в свет как часть программного продукта OLE 2.0 в мае 1993 года.

Корневой интерфейс (тогда он еще не назывался IUnknown) имел в своем составе метод GetClassID. Тот факт, что он был перемещен в IPersist, иллюстрирует принцип поддерживать модель COM настолько простой, насколько это возможно.

В одно время IUnknown не имел метода AddRef. В дальнейшем стало ясно, что запрещение копирования указателей интерфейсов – слишком жесткое ограничение для пользователей.

«Unknown» в «IUnknown» возник как результат создания Тони Вильямсом в декабре 1988 года внутреннего документа фирмы Microsoft под названием ***Object Architecture: Dealing with the Unknown—or—Type Safety in a Dynamically Extensible Class Library*** *(Архитектура объектов: борьба с неизвестным – или – безопасность типов в динамически расширяемой библиотеке классов).*

Решение использовать RPC в качестве механизма межпроцессорного управления (interprocess remoting mechanism) было принято в первые два месяца 1991 года. Служебная записка Боба Аткинсона, озаглавленная ***IAYF Requirements for RPC*** *(Технические требования IAYF для RPC),* документирует требования, предъявленные команде создателей RPC теми, кого впоследствии назвали «командой IAYF». Эта команда отвечала за создание основы

[4] *4* Вспомните речь Билла о IAYF на конгрессе Comdex-1990.

того, что осуществило бы мечту Билла Гейтса об «Information at Your Fingertips». Этой основой и была COM (хотя тогда она еще так не называлась).

Моникеры (monikers) намного мощнее, чем вы думаете.

Это Марк Райланд (Mark Ryland) виноват в том, что некоторые расшифровывают аббревиатуру COM как «Common Object Model» (модель общих объектов). Он глубоко сожалеет об этом и рассыпается в извинениях.

«MEOW» (мяу) в действительности не является сокращением Microsoft Extensible Object Wire (наращиваемый провод для объектов Microsoft). Это шутка Рика (Rick).

Windows NT 3.5 включали в себя первые версии 32-разрядных COM и OLE. Кто-то случайно оставил «#pragma optimization off» в одном из основных заголовочных файлов. Упс! (Oops).

Нет ни одной книги (на английском) о COM, DCOM, OLE или ActiveX, которую бы я не прочитал. Вы, вероятно, найдете мое имя в качестве технического обозревателя в списке разработчиков. Я также сам написал множество статей об этих технологиях, и был первым издателем COM Specification (Спецификация COM). Я провел сотни презентаций COM для технических и нетехнических аудиторий. Из всего этого должно быть ясно, что я потратил огромное количество времени и энергии, чтобы найти наилучший способ объяснить, что такое COM.

А теперь, похоже, весь мой тяжкий труд пропал даром, поскольку после прочтения последнего чернового варианта этой книги мне стало ясно, что никто не объясняет COM лучше, чем Дон Бокс.

Надеюсь, что вы насладитесь этой прогулкой в той же мере, как и я.

Чарли Киндел (Charlie Kindel)
COMовец (COM guy), корпорация Microsoft
Сентябрь 1997 г.

Предисловие Грэйди Буча

Порой о книге можно не просто сказать много хорошего, а сказать это дважды. Это одна из причин, по которой к книге Дона написано два предисловия – она заслуживает этого.

Если вы занимаетесь созданием систем для Windows 95 или NT, вы никак не можете обойтись без COM. Visual Studio, и особенно Visual Basic, скрывают некоторые сложности COM, но если вы: а) действительно хотите понять, что происходит «за кулисами» и/или б) использовать мощность COM, то книга Дона – для вас.

Что мне особенно нравится в этой книге, так это путь, которым идет Дон, освещая COM для читателя. Сначала перед вами открываются проблемы создания рассредоточенных и действующих одновременно систем; затем вам подробно и тщательно объясняют, как эти проблемы решает COM. Даже если вы не знаете абсолютно ничего о COM, когда начинаете читать эту книгу, вас проведут по простой и понятной концептуальной модели COM, после чего вы поймете все задачи, которые COM ставит перед собой, и вам станет ясен характер сил, придающих ему ту структуру и тот образ действий, которыми он обладает. Если же вы – опытный разработчик COM, то вы в полной мере оцените предложенные Доном остроумные и нестандартные способы применения COM для решения обычных задач.

COM – наиболее широко используемая объектная модель для разработки рассредоточенных и действующих одновременно систем. Эта книга поможет вам использовать COM для успешного развития такого рода систем.

Грэйди Буч (Grady Booch)

От автора

Моя работа завершена. Наконец-то я могу отдохнуть, осознав, что наконец изложил на бумаге то, что часто называют *развернутой летописью COM*. Книга отражает эволюцию моего собственного понимания этой норовистой технологии, которую фирма Microsoft в 1993 году сочла достаточно послушной для показа программистскому миру. Хотя я и не присутствовал на конференции профессиональных разработчиков по OLE (OLE Professional Developer's Conference), я по-прежнему чувствую себя так, как будто я занимаюсь COM всегда. После почти четырех лет работы с COM я с трудом вспоминаю доCOMовскую эру программирования. Тем не менее, я прекрасно помню свой мучительный путь через прерию COM в начале 1994 года.

Прошло около шести месяцев, прежде чем я почувствовал, что понял в COM хоть что-либо. В течение этого шестимесячного стартового периода работы с COM я мог успешно писать COM-программы и почти мог объяснить, почему они работают. Однако у меня не было органического понимания того, почему модель программирования COM была тем, чем она была. К счастью, в один из дней, а именно 8 августа 1994 года, примерно через шесть месяцев с момента покупки книги *OLE2 изнутри (Inside OLE2),* на меня снизошло прозрение, и в одночасье COM стал для меня понятен. Это никоим образом не означало, что я понимал каждый интерфейс COM и каждую API-функцию. Но я в значительной степени понял главные побудительные мотивы COM. А значит, стало ясно, как применить эту модель программирования к ежедневным программистским задачам. Многие разработчики испытали нечто похожее. А так как я пишу это введение три августа спустя, эти разработчики все еще вынуждены пройти сквозь этот шестимесячный период ожидания, прежде чем стать продуктивными членами сообщества COM. Я хотел бы надеяться, что моя книга сможет сократить этот период, но обещаний не даю.

Как подчеркивается в этой книге, COM – это в большей степени стиль программирования, чем технология. С этих позиций я стремился не вбивать в читателя подробные описания каждого параметра каждого метода каждого интерфейса. Более того, я старался выделить сущность того, чему в действительности посвящена COM, предоставив документации по SDK заполнить пробелы, остающиеся в каждой главе. Насколько это возможно, я стремился скорее обрисовать те напряжения, которые лежат в основе каждого отдельного аспекта COM, нежели приводить подробные примеры того, как применять каждый интерфейс и каждую API-функцию к какой-нибудь хитроумной иллюстративной программе. Мой собственный опыт показал, что как только я понял *почему*, понимание *как* последовало само собой. И наоборот, простое понимание *как* редко ведет к адекватному проникновению в суть с тем, чтобы экстраполировать за пределы документации. И если кто-то надеется быть в курсе непрерывного развития этой модели программирования, то глубокое понимание ее сути необходимо.

COM является чрезвычайно гибкой основой для создания рассредоточенных объектно-ориентированных систем. Чтобы использовать эту гибкость COM, часто требуется мыслить вне ограничений, диктуемых документацией по SDK, статьями или книгами. Моя личная рекомендация состоит в том, чтобы осознать: все, что вы читаете (в том числе и эта книга), может быть неверным или вопиюще устареть, и вместо этого необходимо сформировать свое собственное понимание этой модели программирования. Безошибочный путь к пониманию этой модели программирования состоит в том, чтобы сконцентрироваться на совершенствовании базового словаря COM. Это может быть достигнуто только через написание программ в стиле COM и анализ того, почему эти программы работают так, как они работают. Чтение книг, статей и документации может помочь, но в конечном счете только выделение времени на обдумывание четырех основных принципов COM (интерфейсы, классы, апартаменты (apartments) и обеспечение безопасности) может повысить вашу эффективность как разработчика COM.

Чтобы помочь разработчику сфокусироваться на этих базовых принципах, я постарался включить в книгу столько кода, сколько это возможно без того, чтобы откровенно снабжать

читателей замысловатыми реализациями для простого копирования их в свой исходный код. А чтобы обеспечить в контексте представительство программной методики COM, в приложения B содержится одно законченное COM-приложение, которое служит примером применения многих концепций, обсуждаемых на протяжении всей этой книги. Кроме того, загружаемый код для этой книги содержит библиотеку кода COM-утилит, которые я счел полезными в моих собственных разработках. Некоторые части этой библиотеки детально обсуждаются в книге, но библиотека в целом включена для демонстрации того, как на деле создавать реализации C++. Заметим также, что большая часть кода, появляющегося в каждой главе, использует макрос assert (объявить) из C-библиотеки этапа выполнения (runtime) с целью подчеркнуть тот факт, что могут встретиться определенные условия «до» и «после». В готовом коде многие из этих операторов assert следует заменить каким-либо кодом, более терпимо обрабатывающим ошибки.

Одним из недостатков издаваемых книг является то, что они часто устаревают уже к моменту их появления на книжных прилавках. И эта книга не исключение. В частности, предстоящий выход в свет COM+ и Windows NT 5.0 несомненно сделают некоторые аспекты этой книги неверными или по крайней мере неполными. Я старался предугадать, какую эволюцию придется претерпеть модели COM из-за выхода Windows NT 5.0, однако в момент написания этого текста Windows NT 5.0 еще не прошла внешнее тестирование, и вся информация подлежит изменениям. COM+ сулит усовершенствовать модель еще дальше; но было, однако, невозможно включить охват COM+ и в то же время выпустить мой манускрипт в этом году. Я настоятельно рекомендую вам изучать как Windows NT 5.0, так и COM+, когда они станут доступны.

Я должен был принять еще одно мучительное решение – как обращаться к различным коммерческим библиотекам, привыкшим реализовывать компоненты COM на C++. Заметив в различных группах новостей Интернета одни и те же проблемы, я предпочел игнорировать ATL (и MSC) и вместо этого сосредоточиться на повседневных темах COM, с которыми должен справляться каждый разработчик независимо от того, какой библиотекой он пользуется. Все больше и больше разработчиков создают спагетти ATL и удивляются, почему ничего не работает. Я твердо уверен, что невозможно выучить COM, программируя в ATL или MSC. Это не значит, что ATL и MSC не являются полезными инструментами для разработки компонентов COM. Это просто означает, что они не годятся для демонстрации или изучения принципов и технологий программирования в COM. Поэтому ATL не подходит для книги, сосредоточенной на модели программирования COM. К счастью, большинство разработчиков находят, что если есть понимание COM, то одолеть основы ATL не составит особого труда.

Наконец, цитаты, которыми начинается каждая глава, – это мой шанс написать для малого раздела книги то, что мне хочется. А чтобы сохранить насколько возможно непрерывность моего изложения, я ограничил свои необузданные и отклоняющиеся от темы сюжеты не более чем 15 строками кода C++ на главу. Обыкновенно этот код/цитата отражает доCOMовский подход к проблеме или концепции, представленной в данной главе. Предлагаю читателю в качестве упражнения попытаться на основе этих намеков реконструировать мое душевное состояние при написании каждой конкретной главы.

Благодарности

Написать книгу невероятно трудно – по крайней мере, для меня. Но я определенно знаю, что два человека страдали больше, чем я, – это моя терпеливая жена Барбара и мой снисходительный сын Макс (который, несмотря на свою юность, предпочитает COM другим объектным моделям). Мои благодарности им обоим: за то, что терпели мое отсутствие и почти постоянное капризное поведение, пока я пытался писать. К счастью, моя только что появившаяся дочь Эван родилась тогда, когда основная часть этой книги была уже написана,

и ее отец стал в достаточной степени и домашним, и приятным. Такие же благодарности – всем сотрудникам DevelopMentor, которые были вынуждены подменять меня, когда я исчезал, чтобы выжать из себя очередную главу.

Большая часть моих ранних размышлений о рассредоточенных системах возникла, когда я в начале 90-х работал на Татсуя Суда (Tatsuya Suda) в университетском колледже в Ирвине. Татсуя учил меня и читать, и писать, и как вести себя с несдержанными пассажирами в токийских поездах. Спасибо и простите.

Благодарю и моего бывшего напарника по офису Дуга Шмидта (Doug Schmidt) – за то, что он представил меня Стэну Липпману (Stan Lippman) из C++ Report. Несмотря на поразительное неприятие Стэном моей первой статьи, мое имя впервые вышло в свет благодаря вам обоим.

Благодарю Майка Хендриксона (Mike Hendrickson) и Алана Фьюэра (Alan Feuer) за то, что поддержали этот проект в самом начале. Спасибо Бену Райану (Ben Ryan) и Джону Уэйту (John Wait) за их терпение. Благодарю Картера Шанклина (Carter Shanklin), который поддерживал этот проект до самого конца.

Спасибо людям из Microsoft Systems Journal, терпевшим мои поздние представления рукописей во время изготовления этой книги. Особые благодарности Джоанне Стэйнхарт (Joanne Steinhart), Гретхен Билсон (Gretchen Bilson), Дэйву Эдсону (Dave Edson), Джо Фланигену (Joe Flanigen), Эрику Маффеи (Eric Maffei), Мишелю Лонгакрэ (Michael Longacre), Джошуа Трупину (Joshua Trupin), Лауре Эйлер (Laura Euler) и Джоан Левинсон (Joan Levinson). Я обещаю больше никогда не запаздывать.

Благодарю Дэвида Чаппела (David Chappell) за то, что он написал лучшую из всех книг по COM. Я искренне рекомендую всем купить экземпляр и прочесть по меньшей мере дважды.

Спасибо приверженцам и фанатикам CORBA и Java, вовлекшим меня в многолетние жаркие сражения на различных конференциях сети Usenet. Ваша неизменная бдительность сделала мое понимание COM неизмеримо более глубоким. Несмотря на то, что я все еще считаю многие ваши аргументы неубедительными и в чем-то даже марсианскими, я уважаю ваше желание выжить.

Некоторые люди в фирме Microsoft очень помогали мне в течение многих лет и прямо или косвенно помогли написать эту книгу. Сара Вильямс (Sara Williams) была первым человеком COM из фирмы Microsoft, с которым я встретился. Сразу объяснив мне, что она недостаточно близко знакома с Биллом, она в утешение тут же представила меня Гретхен Билсон (Gretchen Bilson) и Эрику Маффеи (Eric Maffei) из Microsoft Systems Journal. Сара неизменно была «евангелистом Бокса» внутри фирмы, за что я ей навеки благодарен. Чарли Киндел (Charlie Kindel) написал прелестное предисловие к моей книге, несмотря на плотный график работы и чрезвычайно регулярные визиты к парикмахеру. Нэт Браун (Nat Brown) был первым человеком, показавшим мне, что такое апартаменты (apartments) и непоправимо развративший мой лексикон, засорив его немецким словом «schwing» (вибрировать). Крэйг Брокшмидт (Kraig Brockschmidt) объяснил мне, что один из аспектов COM, выглядящий невероятно изящным, на деле был гротескным хакерским трюком, примененным в последнюю минуту. Дэйв Рид (Dave Reed) представил меня Вайперу (Viper) и выслушивает мои претензии всякий раз, когда я посещаю Рэдмонд. Пэт Хэлланд (Pat Helland) провел целую неделю конференции TechEd'97, вкручивая мне мозги и побуждая меня пересмотреть большинство из моих коренных представлений относительно COM. Скотт Робинсон (Scott Robinson), Андреас Лютер (Andreas Luther), Маркус Хорстман (Markus Horstmann), Мэри Киртланд (Mary Kirtland), Ребекка Норландер (Rebecca Norlander) и Грэг Хоуп (Greg Hope) много сделали для того, чтобы вытащить меня из тьмы. Тэд Хейз (Ted Hase) помогал мне печататься. Рик Хилл (Rick Hill) и Алекс Арманасу (Alex Armanasu) делали большое дело – наблюдали мою спину на техническом фронте. Другие люди из Microsort, оказавшие влияние на мою работу своим участием: Тони Вильямс (Tony Williams), Боб Аткинсон (Bob Atkinson), Крэйг Виттенберг (Craig Wittenberg), Криспин Госвелл (Crispin Goswell), Пол Лич (Paul

Leach), Дэвид Кэйз (David Kays), Джим Спрингфилд (Jim Springfield), Кристиан Бомон (Christian Beaumont), Марио Гёрцел (Mario Goertzel) и Мишель Монтегю (Michael Montague).

Обзор почты DCOM неизменно был для этой книги источником вдохновения и идей. Отдельное спасибо тем, кто прочесывает DCOM для меня: печально известному Марку Райланду (Mark Ryland), *COM-вундеркииду* Майку Нелсону (Mike Nelson), Кэйт Браун (Keith Brown), Тиму Эвалду (Tim Ewald), Крису Селлсу (Chris Sells), Сайджи Эйбрахам (Saji Abraham), Хэнку де Кёнингу (Henk De Koning), Стиву Робинсону (Steve Robinson), Антону фон Штраттену (Anton von Stratten) и Рэнди Путтику (Randy Puttick).

На сюжет этой книги сильно повлияло мое преподавание COM в DevelopMentor в течение нескольких последних лет. Этот сюжет формировался студентами в той же мере, как и моими коллегами-преподавателями. Я мог бы поблагодарить персонально каждого студента. Эддисон Уэсли (Addison Wesley) ограничил авторское предисловие всего лишь двадцатью страницами, я благодарю нынешний состав DevelopMentor, который помог мне отточить мое понимание Essential COM посредством преподавания соответствующего курса и обеспечением бесценной обратной связи: Рона Сумиду (Ron Sumida), Фрица Ониона (Fritz Onion), Скотта Батлера (Scott Butler), Оуэна Толмана (Owen Tallman), Джорджа Шеферда (George Shepherd), Тэда Пэттисона (Ted Pattison), Кейт Браун (Keith Brown), Тима Эвалда (Tim Ewald) и Криса Селлса (Chris Sells). Спасибо вам, ребята! Мои благодарности также Майку Эберкромби (Mike Abercrombie) из DevelopMentor за создание такого окружения, где научный рост участника не сдерживался коммерцией.

Книга могла бы выйти значительно раньше, если бы не Терри Кеннеди (Terry Kennedy) и его друзья из Software AG. Терри был весьма любезен, пригласив меня в Германию помочь им с работой по DCOM/UNIX как раз во время годичного отпуска, который я вырвал специально для написания этой книги. Хотя книга и вышла годом позже из-за того, что я не мог сказать Терри «нет» (это моя вина, а не Терри), но я думаю, что книга получилась несравненно лучше благодаря тому времени, которое я провел за их проектом. В частности, я значительно усилил свою интуицию, работая с Харалдом Стилом (Harald Stiehl), Винни Фролих (Winnie Froehlich), Фолкером Денкхаузом (Volker Denkhaus), Дитмаром Гётнером (Deitmar Gaeitner), Джеффом Ли (Jeff Lee), Дейтером Кеслером (Deiter Kesler), Мартином Кохом (Martin Koch), Блауэром Ауфом (Blauer Aff), Ули Кессом (Uli Kaess), Стивом Уайлдом (Steve Wild) и прославленным Томасом Воглером (Thomas Vogler).

Особые благодарности внимательным читателям, нашедшим ошибки в прежних изданиях этой книги: Тэду Неффу (Ted Neff), Дэну Мойеру (Dan Moyer), Пурушу Рудрекшале (Purush Rudrakshala), Хенгу де Коненгу (Heng de Koneng), Дэйву Хэйлу (Dave Hale), Джорджу Рейли (George Reilly), Стиву Де-Лассусу (Steve DeLassus), Уоррену Янгу (Warren Young), Джеффу Просайзу (Jeff Prosise), Ричарду Граймсу (Richard Grimes), Бэрри Клэвенсу (Barry Klawans), Джеймсу Баумеру (James Bowmer), Стефану Сасу (Stephan Sas), Петеру Заборски (Peter Zaborski), Кристоферу Л. Экерли (Christopher L. Akerley), Роберту Бруксу (Robert Brooks), Джонатану Прайеру (Jonathan Prior), Аллену Чамберсу (Alien Chambers), Тимо Кеттунену (Timo Kettunen), Атулсу Моидекару (Atulx Mohidekar), Крису Хиамсу (Chris Hyams), Максу Рубинштейну (Max Rubinstein), Брэди Хойзингеру (Bradey Honsinger), Санни Томасу (Sunny Thomas), Гарднеру фон Холту (Gardner von Holt) и Тони Вервилосу (Tony Vervilos).

И, наконец, спасибо Шаху Джехану (Shah Jehan) и корпорации «Coca-Cola» за заправку этой затеи горючим в виде производства соответственно превосходной индийской пищи и доступных безалкогольных напитков.

Дон Бокс
Redondo Beach, CA
Август 1997 года
http://www.develop.com/dbox

От издательства

При переводе этой непростой книги о непростой технологии мы попытались сохранить оригинальный авторский стиль, не потеряв при этом ясности изложения. Насколько это удалось, судить читателю.

Редакция выражает особую благодарность Елене Филипповой, руководителю проекта «Королевство Delphi» (http://delphi.vitpc.com), и Артему Артемьеву, ведущему программисту фирмы Data Art, за консультации и помощь при выборе книг для издания.

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства http://www.piter.com .

Все исходные тексты, приведенные в книге, вы найдете по адресу http://www.piter.com/download

Глава 1. COM как улучшенный C++

```
template <class T, class Ex>
class listt : virtual protected CPrivateAlloc
list<T**> mlist;
mutable TWnd mwnd;
virtual ~listt(void);
protected:
explicit listt(int nElems , …);
inline operator unsigned int *(void) const
 return reinterpretcast <int*>(this) ;
template <class X> void clear(X& rx) const throw(Ex);
;
```

Аноним, 1996

C++ уже давно с нами. Сообщество программистов на C++ весьма обширно, и большинство из них хорошо знают о западнях и подводных камнях языка. Язык C++ был создан высоко квалифицированной командой разработчиков, которые, работая в Bell Laboratories, выпустили не только первый программный продукт C++ (CFRONT), но и опубликовали много конструктивных работ о C++. Большинство правил языка C++ было опубликовано в конце 1980-х и начале 1990-х годов. В этот период многие разработчики C++ (включая авторов практически каждой значительной книги по C++) работали на рабочих станциях UNIX и создавали довольно монолитные приложения, использующие технологию компиляции и компоновки того времени. Ясно, что среда, в которой работало это поколение программистов, была в основном создана умами всего сообщества C++.

Одной из главных целей языка C++ являлось позволить программистам строить типы, определенные пользователем (user-defined types – UDTs), которые затем можно было бы использовать вне их исходного контекста. Этот принцип лег в основу идеи создания библиотек классов, или структур, какими мы знаем их сегодня. С момента появления C++ рынок библиотек классов C++ расширялся, хотя и довольно медленно. Одной из причин того. что этот рынок рос не так быстро, как можно было ожидать, был NIH-фактор (not invented here – «изобретен не здесь») среди разработчиков C++. Использовать код других разработчиков часто представляется более трудным, чем воспроизведение собственных наработок. Иногда это представление базируется исключительно на высокомерии разработчика. В других случаях сопротивление использованию чужого кода проистекает из

неизбежности дополнительного умственного усилия, необходимого для понимания чужой идеологии и стиля программирования. Это особенно верно для библиотек-оберток (wrappers), когда необходимо понять не только технологию того, что упаковано, но и дополнительные абстракции, добавленные самой библиотекой.

Другая проблема: многие библиотеки составлены с расчетом на то, что пользователь будет обращаться к исходному коду данной библиотеки как к эталону. Такое повторное использование «белого ящика» часто приводит к огромному количеству связей между программой клиента и библиотекой классов, что с течением времени усиливает неустойчивость всей программы. Эффект чрезмерной связи ослабляет модульный принцип библиотеки классов и усложняет адаптацию к изменениям в реализации основной библиотеки. Это побуждает пользователей относиться к библиотеке как всего лишь к одной из частей исходного кода проекта, а не как к модулю повторного использования. Действительно, разработчики фактически подгоняют коммерческие библиотеки классов под собственные нужды, выпуская «собственную версию», которая лучше приспособлена к данному программному продукту, но уже не является оригинальной библиотекой.

Повторное использование (reuse) кода всегда было одной из классических мотиваций объектного ориентирования. Несмотря на это обстоятельство, написание классов C++, *простых* для повторного использования, довольно затруднительно. Помимо таких препятствий для повторного использования, как этап проектирования (*design-time*) и этап разработки (*development-time*), которые уже можно считать частью культуры C++, существует и довольно большое число препятствий на этапе выполнения (runtime), что делает объектную модель C++ далекой от идеала для создания программных продуктов повторного использования. Многие из этих препятствий обусловлены моделями компиляции и компоновки, принятой в C++. Данная глава будет посвящена техническим проблемам приведения классов C++ к виду компонентов повторного использования. Все задачи будут решаться методами программирования, которые базируются на готовых общедоступных (off-the-shelf) технологиях. В этой главе будет показано, как, применяя эти технологии, можно создать архитектуру для повторного использования модулей, которая позволяла бы динамично и эффективно строить системы из независимо сконструированных двоичных компонентов.

Распространение программного обеспечения и язык C++

Для понимания проблем, связанных с использованием C++ как набора компонентов, полезно проследить, как распространялись библиотеки C++ в конце 1980-х годов. Представим себе разработчика библиотек, который создал алгоритм поиска подстрок за время O [5] (то есть время поиска постоянно, а не пропорционально длине строки). Это, как известно, нетривиальная задача. Для того чтобы сделать алгоритм возможно более простым для пользователя, разработчик должен создать класс строк, основанный на алгоритме, который будет быстро передавать текстовые строки (fast text strings) в любую программу клиента. Чтобы сделать это, разработчику необходимо подготовить заголовочный файл, содержащий определение класса:

```
// faststring.h
class FastString

char *mpsz;
```

[5] *1* В момент написания этого текста автор не имел работающей версии этого алгоритма, годной для публикации. Детали такой реализации оставлены как упражнение для читателя.

```
public:
FastString(const char *psz);
~FastString(void);
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
//возвращает смещение
;
```

После того как класс определен, разработчик должен реализовать его функции-члены в отдельном файле:

```
// FastString.cpp
#include «faststring.h»
#include <string.h>
FastString::FastString(const char *psz) : mpsz(new char [strlen(psz) + 1])
 strcpy(mpsz, psz);
FastString::~FastString(void)
 delete[] mpsz;
int FastString::Length(void) const
 return strlen(mpsz);
int FastString::Find(const char *psz) const

//O(1) lookup code deleted for> clarity
1
// код поиска 0(1) удален для ясности
```

Библиотеки C++ традиционно распространялись в форме исходного кода. Ожидалось, что пользователи библиотеки будут добавлять реализации исходных файлов и создаваемую ими систему и перекомпилировать библиотечные исходные файлы на месте, с использованием своего компилятора C++. Если предположить, что библиотека написана на наиболее употребительной версии языка C++, то такой подход был бы вполне работоспособным. Подводным камнем этой схемы было то, что исполняемый код этой библиотеки должен был включаться во все клиентские приложения.

Предположим, что для показанного выше класса *FastString* сгенерированный машинный код для четырех методов занял 16 Мбайт пространства в результирующем исполняемом файле. Напомним, что при выполнении O(1)-поиска может потребоваться много пространства для кода, чтобы обеспечить заданное время исполнения, – дилемма, которая ограничивает большинство алгоритмов. Как показано на рис. 1.1, если три приложения используют библиотеку *FastString* , то каждая из трех исполняемых программ будет включать в себя по 16 Мбайт кода. Это означает, что если конечный пользователь инсталлирует все три клиентских приложения, то реализация *FastString* займет 48 Мбайт дискового пространства. Хуже того – если конечный пользователь запустит все три клиентских приложения одновременно, то код *FastString* займет 48 Мбайт виртуальной памяти, так как операционная система не может обнаружить дублирующий код, имеющийся в каждой исполняемой программе.

Рис. 1.1 Три клиента FastString

Есть еще одна проблема в таком сценарии: когда разработчик библиотеки находит дефект в классе *FastString* , нет способа всюду заменить его реализацию. После того как код *FastString* скомпонован с клиентским приложением, невозможно исправить машинный код *FastString* непосредственно в компьютере конечного пользователя. Вместо этого разработчик библиотеки должен известить разработчиков каждого клиентского приложения об изменениях в исходном коде и надеяться, что они переделают свои приложения, чтобы получить эффект от этих исправлений. Ясно, что модульность компонента *FastString* утрачивается, как только клиент запускает компоновщик и заново формирует исполняемый файл.

Динамическая компоновка и C++

Один из путей решения этих проблем – упаковка класса FastString в динамически подключаемую библиотеку (Dynamic Link Library – DLL). Это может быть сделано несколькими способами. Простейший из них – использовать директиву компилятора, действующую на уровне классов, чтобы заставить все методы FastString экспортироваться из DLL. Компилятор Microsoft C++ предусматривает для этого ключевое слово _declspec(dllexport):

```
class _declspec(dllexport) FastString

char *m_psz;
public:
FastString(const char *psz);
~FastString(void);
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение
;
```

В этом случае все методы *FastString* будут добавлены в список экспорта соответствующей библиотеки DLL, что позволит записать время выполнения каждого метода в его адрес в памяти. Кроме того, компоновщик создаст библиотеку импорта (import library), которая объявляет символы для методов *FastString* . Вместо того чтобы содержать сам код, библиотека импорта включает в себя ссылки на имя файла DLL и имена

экспортируемых символов. Когда клиент обращается к библиотеке импорта, эти ссылки добавляются к исполняемой программе. Это побуждает загрузчик динамически загружать DLL *FastString* во время выполнения и размещать импортируемые символы в соответствующие ячейки памяти. Это размещение автоматически происходит в момент запуска клиентской программы операционной системой.

Рисунок 1.2 иллюстрирует модель *FastString* на этапе выполнения (runtime model), объявляемую из DLL. Заметим, что библиотека импорта достаточно мала (примерно вдвое больше, чем суммарный размер экспортируемого символьного текста). Когда класс экспортируется из DLL, код *FastString* должен присутствовать на жестком диске пользователя только один раз. Если даже несколько клиентов применяют этот код для своей библиотеки, загрузчик операционной системы обладает достаточным интеллектом, чтобы разделить физические страницы памяти, содержащие исполняемый код *FastString* (только для чтения), между всеми клиентскими программами. Кроме того, если разработчик библиотеки найдет дефект в исходном коде, теоретически возможно послать новую DLL конечному пользователю, исправляя дефектную реализацию для всех клиентских приложений *сразу*. Ясно, что перемещение библиотеки *FastString* в DLL является важным шагом на пути превращения класса C++ в заменяемый и эффективный компонент повторного использования.

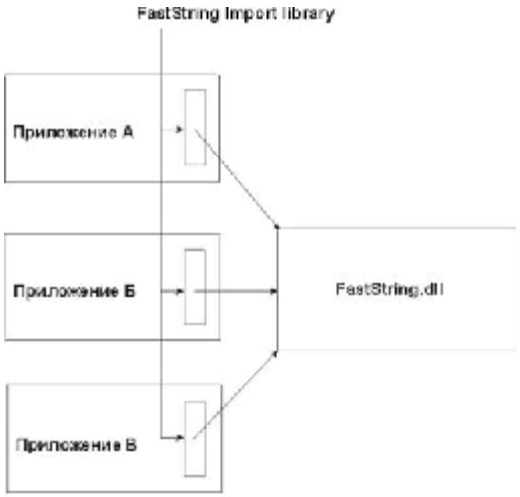

Рис. 1.2 FastString как DLL

C++ и мобильность

Поскольку вы решили распространять классы C++ как DLL, вы непременно столкнетесь с одним из фундаментальных недостатков C++ – недостаточной стандартизацией на двоичном уровне. Хотя рабочий документ *ISO/ANSI C++ Draft Working Paper (DWP)* предпринимает попытку определить, какие программы будут транслироваться и каковы будут семантические эффекты при их запуске, двоичная динамическая модель C++ ею не стандартизируется. Впервые клиент сталкивается с этой проблемой при попытке скомпоновать библиотеку импорта DLL *FastString* из среды развития C++, отличной от той, в которой он привык строить эту DLL.

Для обеспечения перегрузки операторов и функций компиляторы C++ обычно видоизменяют символическое имя каждой точки входа, чтобы разрешить многократное использование одного и того же имени (или с различными типами аргументов, или в различных областях действия) без нарушения работы существующих компоновщиков для языка C. Этот прием часто называют *коррекцией имени*. Несмотря на то что ARM (C++ Annotated Reference Manual) документировала схему кодирования, использующуюся в CFRONT, многие разработчики трансляторов предпочли создать свою собственную схему

коррекции. Поскольку библиотека импорта *FastString* и DLL экспортирует символы, используя корректирующую схему того транслятора, который создал DLL (то есть GNU C++), клиенты, скомпилированные другим транслятором (например, Borland C++), не могут быть корректно скомпонованы с библиотекой импорта. Классическая методика использования *extern "C"* для отключения коррекции символов не поможет в данном случае, так как DLL экспортирует функции-члены (методы), а не глобальные функции.

Для решения этой проблемы можно проделать фокусы с клиентским компоновщиком, применяя файл описания модуля *(Module Definition File)* , известный как DEF-файл. Одно из свойств DEF-файлов заключается в том, что они позволяют экспортируемым символам совмещаться с различными импортируемыми символами. Имея достаточно времени и информации относительно каждой схемы коррекции, разработчик библиотек может создать особую библиотеку импорта для каждого компилятора. Это утомительно, но зато позволяет любому компилятору обеспечить совместимость с DLL на уровне компоновки, при условии, что разработчик библиотеки заранее ожидал ее использование и создал нужный DEF-файл.

Если вы разрешили проблемы, возникшие при компоновке, вам еще придется столкнуться с более сложными проблемами несовместимости, которые связаны со сгенерированным кодом. За исключением простейших языковых конструкций, разработчики трансляторов часто предпочитают реализовывать особенности языка своими собственными путями. Это формирует объекты, недоступные для кода, созданного любым другим компилятором. Классическим примером таких языковых особенностей являются исключительные ситуации (исключения). Исключительная ситуация в среде C++, исходящая от функции, которая была транслирована компилятором Microsoft, не может быть надежно перехвачена клиентской программой, оттранслированной компилятором Watcom. Это происходит потому, что DWP не может определить, как должна выглядеть та или иная особенность языка на этапе выполнения, поэтому для каждого разработчика компилятора вполне естественно реализовать такую языковую особенность в своей собственной, новаторской манере. Это несущественно при построении независимой однобинарной (single-binary) исполняемой программы, так как весь код будет транслироваться и компоноваться в одной и той же среде. При построении мультибинарных (multibinary) исполняемых программ, основанных на компонентах (component-based), это представляет серьезную проблему, так как каждый компонент может, очевидно, быть построен с использованием другого компилятора и компоновщика. Отсутствие двоичного стандарта в C++ ограничивает возможности того, какие особенности языка могут быть использованы вне границ DLL. Это означает, что простой экспорт функций-членов C++ из DLL недостаточен для создания независимого от разработчика набора компонентов.

Инкапсуляция и C++

Предположим, что вам удалось преодолеть проблемы с транслятором и компоновщиком, описанные в предыдущем разделе. Очередное препятствие при построении двоичных компонентов на C++ появится, когда вы будете проводить инкапсуляцию (encapsulation), то есть формирование пакета. Посмотрим, что получится, если организация, использующая *FastString* в приложении, возьмется выполнить невыполнимое: закончит разработку и тестирование за два месяца до срока рассылки продукта. Пусть также в течение этих двух месяцев некоторые из наиболее скептически настроенных разработчиков решили протестировать *O(1)* -поисковый алгоритм *FastString* , запустив профайлер своего приложения. К их большому удивлению, *FastString::Find* стала бы на самом деле работать очень быстро, независимо от заданной длины строки. Однако с оператором *Length* дело обстоит не столь хорошо, так как *FastString::Length* использует подпрограмму *strlen* из динамической библиотеки C. Эта подпрограмма – алгоритм *O(n)* – осуществляет линейный поиск по строкам с использованием символа конца строки (null terminator); скорость его

работы пропорциональна длине строки. Столкнувшись с тем, что клиентское приложение может многократно вызывать оператор *Length* , один из таких скептиков, скорее всего, свяжется с разработчиком библиотеки и попросит его убыстрить *Length* , чтобы его работа также не зависела от длины строки. Но здесь есть одно препятствие. Разработчик библиотеки уже закончил свою разработку и, скорее всего, не расположен менять одну строку исходного кода, чтобы воспользоваться преимуществами улучшенного метода *Length* . Кроме того, некоторые другие разработчики, возможно, уже выпустили свои продукты, основанные на текущей версии *FastString* , и теперь разработчик библиотеки не имеет морального права изменять эти приложення.

С этой точки зрения нужно просто вернуться к определению класса *FastString* и решить, что можно изменить и что необходимо сохранить, чтобы уже установленная база успешно функционировала. К счастью, класс *FastString* был разработан с учетом возможности инкапсуляции, и все его элементы данных (*data members*) являются закрытыми (*private*). Это придает классу значительную гибкость, так как ни одна клиентская программа не может непосредственно получить доступ к элементам данных *FastString* . В силу того, что по отношению к четырем открытым (*public*) членам класса не было сделано никаких изменений, то и в любом клиентском приложении никаких изменений также не потребуется. Вооружившись этой верой, разработчик библиотеки переходит к реализации *FastString* версии 2.0.

Очевидным улучшением является следующее решение: в тексте конструктора (*constructor*) занести длину строки в кэш и возвращать кэшированную длину в новой версии метода *Length* . Так как строка не может быть изменена после создания, нет необходимости беспокоиться, что ее длина будет вычисляться многократно. В действительности длина уже однажды вычислена в конструкторе при назначении буфера, так что понадобится только горстка дополнительных машинных инструкций. Вот каким будет модифицированное определение класса:

```
// faststring.h version 2.0
class declspec(dllexport) FastString
const int mcch;
// count of characters
// число символов
char mpsz;
public:
FastString(const char *psz);
~FastString(void);
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset – возвращает смещение
;
```

Отметим, что единственной модификацией является добавление закрытого элемента данных. Чтобы правильно инициализировать такой элемент, конструктор должен быть изменен следующим образом:

```
FastString::FastString(const char *psz) : mcch(strlen(psz)), mpsz(new char[mcch + 1])

strcpy(mpsz, psz);
```

С введением кэшированной длины метод Length становится тривиальным:

```
int FastString::Length(void) const

return mcch;
// return cached length
// возвращает скрытую длину
```

Сделав эти три модификации, разработчик библиотеки может теперь перестроить DLL *FastString* и сопутствующий ей набор тестов, которые полностью проверяют каждый аспект класса *FastString* . Разработчик будет приятно удивлен, узнав, что принцип инкапсуляции обошелся ему дешево, и в исходных текстах тестов не понадобилось делать никаких изменений. После проверки того. что новая DLL работает правильно, разработчик библиотек отсылает *FastString* версии 2.0 клиенту, будучи уверенным, что вся работа завершена.

Когда клиенты, заказавшие изменения, получают модернизированный *FastString* , они включают новое определение класса и DLL в систему контроля своего исходного кода и запускают тестирование нового и улучшенного *FastString* . Подобно разработчику библиотеки, они тоже приятно удивлены: для того, чтобы воспользоваться преимуществами новой версии *Length* , не требуется никаких модификаций исходного кода. Вдохновленная этим опытом, команда разработчиков убеждает начальство включить новую DLL в окончательный «золотой» CD, уже готовый для выпуска. Это тот редкий случай, когда руководство идет навстречу энтузиастам-разработчикам и включает в окончательный продукт новую DLL. Подобно большинству программ инсталляции, описание установки клиентской программы настроено на молчаливое (без предупреждения) замещение всех старых версий *FastString* DLL, какие есть на машине конечного пользователя. Это выглядит вполне безобидно, поскольку эти изменения не затронули открытый интерфейс класса, так что тотальная молчаливая модернизация под версию 2.0 *FastString* только улучшит любые имеющиеся клиентские приложения, которые были установлены раньше.

Представим себе следующий сценарий: конечные пользователи наконец-то получают свои экземпляры вожделенного продукта. Каждый из них тут же бросает все и устанавливает новое приложение на свою машину, дабы попробовать его. После того как высохли слезы восторга от того, что наконец-то можно делать быстрый текстовый поиск, пользователь возвращается к его или ее нормальному состоянию и запускает ранее установленное приложение, которое также имеет неосторожность использовать DLL *FastString* . Первые несколько минут всё идет хорошо. Затем внезапно появляется сообщение, что возникла исключительная ситуация и что вся работа конечного пользователя пропала. Он пытается запустить приложение снова, но на этот раз диалоговое окно об исключительной ситуации появляется почти сразу. Конечный пользователь, привычный к употреблению современного программного обеспечения, переустанавливает операционную систему и все приложения, но даже это не спасает от повторения исключительной ситуации. Что же произошло?

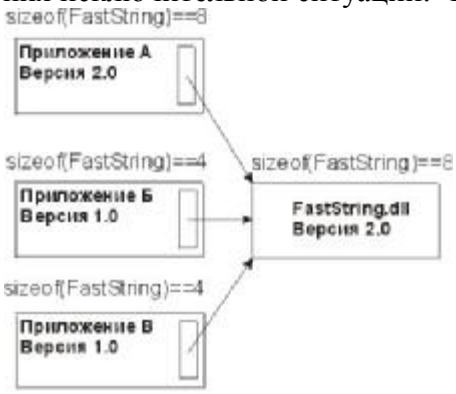

Рис. 1.3. C++ и инкапсуляция

А произошло то, что разработчик библиотеки был убаюкан верой в то, что C++ поддерживает инкапсуляцию. Хотя C++ и поддерживает *синтаксическую* инкапсуляцию через свои закрытые и защищенные ключевые слова, в стандарте C++ ничего не сказано о *двоичной* инкапсуляции. Это происходит потому, что модель трансляции C++ требует, чтобы клиентский компилятор имел доступ ко всей информации относительно двоичного представления объектов, – с целью обработать экземпляр класса или делать невиртуальные вызовы метода. Это включает в себя информацию о размере и порядке закрытых и защищенных элементов данных объекта. Рассмотрим сценарий, показанный на рис. 1.3. Версия 1.0 *FastString* требует четыре байта на экземпляр (принимая *sizeof(char *) == 4*). Клиенты написанного под версию 1.0 определения класса выделяют четыре байта памяти под вызов конструктора класса. Конструктор, деструктор и методы версии 2.0 (а именно эти версии содержатся в DLL в машине конечного пользователя) ожидают, что клиент выделил восемь байт на экземпляр (принято *sizeof(int) == 8*), и не предусматривают собственных резервов для записи во все восемь байт. К сожалению, у клиентов с версией 1.0 вторые четыре байта этого объекта на самом деле принадлежат кому-то другому, и запись в это место указателя на текстовую строку недопустима, о чем и сообщает диалог исключительной ситуации.

Существует общее решение проблемы версий – переименовывать DLL всякий раз, когда появляется новая версия. Такая стратегия принята в Microsoft Foundation Classes (MFC). Когда номер версии включен в имя файла DLL (например, *FastString10.DLL* , *FastString20.DLL*), клиенты всегда загружают ту версию DLL, с которой они были сконфигурированы, независимо от присутствия в системе других версий. К сожалению, со временем, из-за недостаточного опыта в системном конфигурировании, число версий DLL, имеющихся в системе конечного пользователя, может превысить реальное число пользовательских приложений. Чтобы убедиться в этом, достаточно проверить системный каталог любого компьютера, проработавшего больше шести месяцев.

В конечном счете, проблема управления версиями коренится в модели трансляции C++, не рассчитанной на поддержку независимых двоичных компонентов. Требуя знания клиентом двоичного представления объектов, C++ предполагает тесную двоичную связь между клиентом и исполняемыми программами объекта. Обычно такая связь является преимуществом C++, так как она позволяет трансляторам генерировать весьма эффективный код. К сожалению, эта тесная двоичная связь не позволяет переместить реализации класса без проведения клиентом повторной компиляции. По причине этой связи и несовместимости транслятора и компоновщика, упомянутых в предыдущем разделе, простой экспорт определений класса C++ из DLL *не* обеспечивает приемлемой архитектуры двоичных компонентов.

Отделение интерфейса от реализации

Концепция инкапсуляции основана на разделении того, как объект выглядит (его интерфейса), и того, как он в действительности работает (его реализации). Проблема в C++ в том, что этот принцип неприменим на двоичном уровне, так как класс C++ одновременно является и интерфейсом, и реализацией. Этот недостаток может быть преодолен, если смоделировать две новые абстракции, являющиеся классами C++, но различающиеся по своей сущности. Если определить один класс C++ как интерфейс для типа данных, а второй – как саму реализацию типа данных, то конструктор объектов теоретически может модифицировать некоторые детали класса реализации, в то время как класс интерфейса останется неизменным. Все, что нужно, – это выдержать соотношение интерфейса с его реализацией так, чтобы не показывать клиенту никаких деталей реализации.

Класс интерфейса должен содержать только такое описание основных типов данных, какое должен, по мнению разработчика, представлять себе клиент. Поскольку интерфейс не

должен сообщать ни о каких деталях реализации, класс интерфейса C++ не может содержать никаких элементов данных, которые могут быть использованы в реализации объекта. Вместо этого класс интерфейса должен содержать только описания методов для каждой открытой операции объекта. Класс реализации C++ будет содержать фактические элементы данных, необходимые для обеспечения функционирования объекта. Одним из простейших подходов является использование класса-дескриптора (handle-class) в качестве интерфейса. Класс-дескриптор мог бы просто содержать непрозрачный (opaque) указатель, чей тип никогда не может быть полностью определен клиентом. Следующее определение класса демонстрирует эту технику:

```
// FastStringItf.h
class declspec(dllexport) FastStringItf

class FastString;
// introduce name of impl. class
// вводится имя класса реализации
FastString *mpThis;
// opaque pointer (size remains constant)
// непрозрачный указатель (размер остается постоянным)
public: FastStringItf(const char *psz);
~FastStringItf(void);
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение
;
```

Заметим, что двоичное представление этого класса интерфейса не меняется с добавлением или удалением элементов данных из класса реализации ***FastString*** . Кроме того, использование опережающего объявления означает, что определение класса ***FastString*** не является необходимым для трансляции этого заголовочного файла. Это эффективно скрывает все детали реализации ***FastString*** от транслятора клиента. При использовании этого способа машинный код для методов интерфейса становится единственной точкой входа в DLL объекта, и их двоичные сигнатуры никогда не изменятся. Реализации методов класса интерфейса просто передают вызовы методов действующему классу реализации:

```
// faststringitf.cpp
// (part of DLL, not client)
// (часть DLL, а не клиента)
#include «faststring.h»
#include «faststringitf.h»
FastStringItf::FastStringItf(const char *psz) : mpThis(new FastString(psz))
 assert(mpThis != 0);
FastStringItf::~FastStringItf(vo1d)
 delete mpThis;
int FastStringItf::Length(void) const
 return mpThis->Length();
int FastStringItf::Find(const char *psz) const
 return mpThis->Find(psz);
```

Эти передающие методы должны быть транслированы как часть DLL *FastString* , так что когда двоичное представление класса реализации *FastString* меняется, вызов нового оператора в конструкторе *FastStringItf* будет сразу же перекомпилирован, если, конечно, зарезервировано достаточно памяти. И опять клиент не получит описания *класса реализации FastString* . Это дает разработчику *FastString* возможность со временем развивать реализацию без прерывания существующих клиентов.

Рис. 1.4 Классы-дескрипторы в качестве интерфейсов

Рисунок 1.4 показывает, как использовать классы-дескрипторы для отделения интерфейса от реализации на этапе выполнения. Заметим, что косвенный подход, введенный классом интерфейса, устанавливает двоичную защитную стену (firewall – брандмауэр) между клиентом и реализацией объекта. Эта двоичная стена очень точно описывает, как клиент может сообщаться с реализацией. Все связи клиент-объект осуществляются через класс интерфейса, который содержит очень простой двоичный протокол для входа в область реализации объекта. Этот протокол не содержит никаких деталей класса реализации в C++.

Хотя методика использования классов-дескрипторов имеет свои преимущества и безусловно приближает нас к возможности безопасного извлечения классов из DLL, она также имеет свои недостатки. Отметим, что класс интерфейса вынужден явно передавать каждый вызов метода классу реализации. Для простого класса вроде *FastString* только с двумя открытыми операторами, конструктором и деструктором, это не проблема. Для большой библиотеки классов с сотнями или тысячами методов написание этих передающих процедур было бы весьма утомительным и явилось бы потенциальным источником ошибок. Кроме того, для областей с повышенными требованиями к эффективности программ (performance-critical domains), цена двух вызовов для каждого метода (один вызов на интерфейс, один вложенный вызов на реализацию) весьма высока. Наконец, методика классов-дескрипторов не полностью решает проблемы совместимости транслятора/компоновщика, а они все же должны быть решены, если мы хотим иметь основу, действительно пригодную для создания компонентов повторного использования.

Абстрактные базы как двоичные интерфейсы

Оказывается, применение техники разделения интерфейса и реализации может решить и проблемы совместимости транслятора/компоновщика C++. При этом, однако, определение класса интерфейса должно принять несколько иную форму. Как отмечалось ранее, проблемы совместимости возникают из-за того, что разные трансляторы имеют различные соображения по поводу того, как

1. передавать особенности языка на этапе выполнения;
2. символические имена будут представлены на этапе компоновки.

Если бы кто-нибудь придумал, как скрыть детали реализации транслятора/компоновщика за каким-либо двоичным интерфейсом, это сделало бы написанные на C++ библиотеки DLL значительно более широко используемыми.

Двоичная защита, то есть тот факт, что класс интерфейса C++ не использует языковых конструкций, зависящих от транслятора, решает проблему зависимости от транслятора/компоновщика. Чтобы сделать эту независимость более полной, необходимо в первую очередь определить те аспекты языка, которые имеют одинаковую реализацию в

разных трансляторах. Конечно, представление на этапе выполнения таких сложных типов, как C-структуры (structs), может быть выдержано инвариантным по отношению к трансляторам. Это – основное, что должен делать системный интерфейс, основанный на C, и иногда это достигается применением условно транслируемых определений типа прагм (pragmas) или других директив транслятора. Второе, что следует сделать, – это заставить все компиляторы проходить параметры функций в одном и том же порядке (слева направо, справа налево) и зачищать стек также одинаково. Подобно совместимости структур, это также решаемая задача, и для унификации работы со стеком часто используются условные директивы транслятора. В качестве примера можно привести макросы *WINAPI/WINBASEAPI* из Win32 API. Каждая извлеченная из системных DLL функция определена с помощью этих макросов:

```
WINBASEAPI void WINAPI Sleep(DWORD dwMsecs);
```

Каждый разработчик транслятора определяет эти символы препроцессора для создания гибких стековых фреймов. Хотя в среде производителей может возникнуть желание использовать аналогичную методику для определений всех методов, фрагменты программ в этой главе для большей наглядности ее не используют.

Третье требование к независимости трансляторов – наиболее уязвимое для критики из всех, так как оно делает возможным определение двоичного интерфейса: все трансляторы C++ с заданной платформой одинаково осуществляют механизм вызова виртуальных функций. Действительно, это требование единообразия применимо только к классам, не имеющим элементов данных, а имеющим не более одного базового класса, который также не имеет элементов данных. Вот что означает это требование для следующего простого определения класса:

```
class calculator

public: virtual void add1(short x);
virtual void add2(short x, short y);
;
```

Все трансляторы с данной платформой должны создать эквивалентные последовательности машинного кода для следующего фрагмента программы пользователя:

```
extern calculator *pcalc;
pcalc->add1(1);
pcalc->add2(1, 2);
```

Отметим, что требуется не *идентичность* машинного кода на всех трансляторах, а его *эквивалентность*. Это означает, что каждый транслятор должен делать одинаковые допущения относительно того, как объект такого класса размещен в памяти и как его виртуальные функции динамически вызываются на этапе выполнения.

Впрочем, это не такое уж блестящее решение проблемы, как может показаться. Реализация виртуальных функций на C++ на этапе выполнения выливается в создание конструкций *vptr* и *vtbl* практически на всех трансляторах. При этой методике транслятор молча генерирует статический массив указателей функций для каждого класса, содержащего виртуальные функции. Этот массив называется *vtbl* (virtual function table – таблица виртуальных функций) и содержит один указатель функции для каждой виртуальной функции, определенной в данном классе или в ее базовом классе. Каждый объект класса содержит единственный невидимый элемент данных, именуемый *vptr* (virtual function pointer – указатель виртуальных функций); он автоматически инициализируется конструктором для

указания на таблицу vtbl класса. Когда клиент вызывает виртуальную функцию, транслятор генерирует код, чтобы разыменовать указатель *vptr* , занести его в *vtbl* и вызвать функцию через ее указатель, найденный в назначенном месте. Так на C++ обеспечивается полиморфизм и диспетчеризация динамических вызовов. Рисунок 1.5 показывает представление на этапе выполнения массивов *vptr/vtbl* для класса *calculator* , рассмотренного выше.

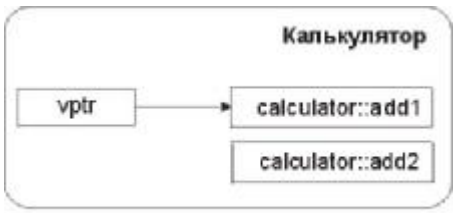

Рис. 1.5 Размещение vptr/vtbl

Фактически каждый действующий в настоящее время качественный транслятор C++ использует базовые концепции *vprt* и *vtbl* . Существует два основных способа размещения таблицы *vtbl* : с помощью *CFRONT* и корректирующего переходника (*adjuster thunk*). Каждый из этих приемов имеет свой способ обращения с тонкостями множественного наследования. К счастью, на каждой из имеющихся платформ доминирует один из способов (трансляторы Win32 используют *adjuster thunk* , Solaris – стиль *CFRONT* для *vtbl*). К тому же формат таблицы *vtbl* не влияет на исходный код C++, который пишет программист, а скорее является артефактом сгенерированного кода. Желающие узнать подробности об этих двух способах могут обратиться к прекрасной книге Стэна Липпмана «Объектная модель C++ изнутри» (Stan Lippman. Inside C++ Object Model).

Основываясь на столь далеко идущих допущениях, теперь можно решить проблему зависимости от транслятора. Предполагая, что все трансляторы на данной платформе одинаково реализуют механизм вызова виртуальной функции, можно определить класс интерфейса C++ так, чтобы глобальные операции над типами данных определялись в нем как виртуальные функции; тогда можно быть уверенным, что все трансляторы будут генерировать эквивалентный машинный код для вызова методов со стороны клиента. Это предположение об единообразии означает, что ни один класс интерфейса не имеет элементов данных и ни один класс интерфейса не может быть прямым потомком более чем одного класса интерфейса. Поскольку в классе интерфейса нет элементов данных, эти методы практически невозможно использовать.

Чтобы подчеркнуть это обстоятельство, полезно определить члены интерфейса как простые виртуальные функции, указав, что класс интерфейса задает только возможность вызова методов, а не их реализацию.

```
// ifaststring.h
class IFastString

public: virtual int Length(void) const = 0;
virtual int Find(const char *psz) const = 0;
;
```

Определение этих методов как чисто виртуальных также дает знать транслятору, что от класса интерфейса не требуется никакой реализации этих методов. Когда транслятор генерирует таблицу vtbl для класса интерфейса, входная точка для каждой простой виртуальной функции является или нулевой (null), или точкой входа в C-процедуру этапа выполнения (_purecall в Microsoft C++), которая при вызове генерирует логическое утверждение. Если бы метод не был определен как чисто виртуальный, транслятор попытался бы включить в соответствующую входную точку vtbl системную реализацию метода класса интерфейса, которая в действительности не существует. Это вызвало бы ошибку компоновки. Определенный таким образом класс интерфейса является абстрактным

базовым классом. Соответствующий класс реализации должен порождаться классом интерфейса и перекрывать все чисто виртуальные функции содержательными реализациями. Эта наследственная связь проявится в объектах, которые в качестве своего представления имеют двоичное надмножество представления класса интерфейса (которое как раз и есть vptr/vtbl). Дело в том, что отношение «является» («is-a») между порождаемым и базовым классами применяется на двоичном уровне в C++ так же, как и на уровне моделирования в объектно-ориентированной разработке:

```
class FastString : public IFastString

const int m_cch;
// count of characters
// число символов
char *m_psz;
public:
FastString(const char *psz);
~FastString(void);
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение
;
```

Поскольку ***FastString*** порождается от ***IFastString*** , двоичное представление объектов ***FastString*** должно быть надмножеством двоичного представления ***IFastString*** . Это означает, что объекты ***FastString*** будут содержать указатель ***vptr*** , указывающий на совместимую с таблицей ***vtbl IFastString*** . Поскольку классу ***FastString*** можно приписывать различные конкретные типы данных, его таблица ***vtbl*** будет содержать указатели на существующие реализации методов ***Length*** и ***Find*** . Их связь показана на рис. 1.6.

Даже несмотря на то, что открытые операторы над типами данных подняты до уровня чисто виртуальных функций в классе интерфейса, клиент не может приписывать значения объектам ***FastString*** , не имея определения класса для класса реализации. При демонстрации клиенту определения класса реализации от него будет скрыта двоичная инкапсуляция интерфейса; что не позволит клиенту использовать класс интерфейса. Одним из разумных способов обеспечить клиенту возможность использовать объекты ***FastString*** является экспорт из DLL глобальной функции, которая будет вызывать новый оператор от имени клиента. При условии, что эта подпрограмма экспортируется с опцией ***extern "C"*** , она будет доступна для любого транслятора C++.

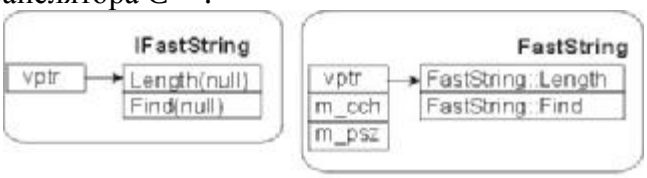

Рис. 1.6. Двоичное представление классов интерфейс/реализация

```
// ifaststring.h
class IFastString
public:
virtual int Length(void) const = 0;
virtual int Find(const char *psz) const = 0;
;
```

```
extern "C"
IFastString *CreateFastString(const char *psz);
// faststring.cpp (part of DLL)
// faststring.cpp (часть DLL)
IFastString *CreateFastString (const char *psz)
 return new FastString(psz);
```

Как было в случае класса-дескриптора, новый оператор вызывается исключительно внутри DLL ***FastString*** , а это означает, что размер и расположение объекта будут установлены с использованием того же транслятора, который транслировал все методы реализации.

Последнее препятствие, которое предстоит преодолеть, относится к уничтожению объекта. Следующая клиентская программа пройдет трансляцию, но результаты будут непредсказуемыми:

```
int f(void)

IFastString *pfs = CreateFastString(«Deface me»);
int n = pfs->Find(«ace me»);
delete pfs;
return n;
```

Непредсказуемое поведение вызвано тем фактом, что деструктор класса интерфейса не является виртуальным. Это означает, что вызов оператора ***delete*** не сможет динамически найти последний порожденный деструктор и рекурсивно уничтожит объект ближайшего внешнего типа по отношению к базовому типу. Поскольку деструктор ***FastString*** никогда не вызывается, в данном примере из буфера исчезнет строка «Deface me», которая должна там присутствовать.

Очевидное решение этой проблемы – сделать деструктор виртуальным в классе интерфейса. К сожалению, это нарушит независимость класса интерфейса от транслятора, так как положение виртуального деструктора в таблице ***vtbl*** может изменяться от транслятора к транслятору. Одним из конструктивных решений этой проблемы является добавление к интерфейсу явного метода ***Delete*** как еще одной чисто виртуальной функции, чтобы заставить производный класс уничтожать самого себя в своей реализации этого метода. В результате этого будет выполнен нужный деструктор. Модифицированная версия заголовочного файла интерфейса выглядит так:

```
// ifaststring.h
class IFastString

public:
virtual void Delete(void) = 0;
virtual int Length(void) const = 0;
virtual int Find(const char *psz) const = 0;
;
extern "C"
IFastString *CreateFastString (const char *psz);
```

она влечет за собой соответствующее определение класса реализации:

```
// faststring.h
```

```
#include «ifaststring.h»
class FastString : public IFastString
 const int mcch;
// count of characters
// счетчик символов
char *mpsz; public: FastString(const char *psz);
~FastString(void);
void Delete(void);
// deletes this instance
// уничтожает этот экземпляр
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение
;
// faststring.cpp
#include <string.h>
#include «faststring.h»
IFastString* CreateFastString (const char *psz)
return new FastString(psz);

FastString::FastString(const char *psz) : mcch(strlen(psz)) , mpsz(new char[m cch + 1])
strcpy(mpsz , psz);

void FastString::Delete(void)
delete this;

FastString::~FastString(void)
delete[] mpsz;

int FastString::Lengtn(void) const
return mcch;

int FastString::Find(const char *psz) const
// O(1) lookup code deleted for clarity
// код поиска 0(1) уничтожен для ясности
```

Рисунок 1.7 показывает представление *FastString* на этапе выполнения. Чтобы использовать тип данных *FastString* , клиентам надо просто включить в программу файл определения интерфейса и вызвать *CreateFastString* :

```
#include «ifaststring.h»
int f(void)
 int n = -1;
IFastString *pfs = CreateFastString(«Hi Bob!»);
if (pfs)  n = pfs->Find(«ob»);
pfs->Delete();
return n;
```

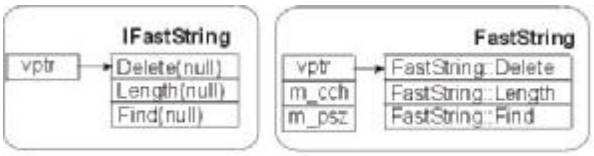

Рис. 1.7. FastString, использующий абстрактные базы в качестве интерфейсов

Отметим, что все, кроме одной, точки входа в DLL *FastString* являются виртуальными функциями. Виртуальные функции класса интерфейса всегда вызываются косвенно, через указатель функции, хранящийся в таблице *vtbl* , избавляя клиента от необходимости указывать их символические имена на этапе разработки. Это означает, что методы интерфейса защищены от различий в коррекции символических имен на разных трансляторах. Единственная точка входа, которая явно компонуется по имени, – это *CreateFastString* – глобальная функция, которая обеспечивает клиенту доступ в мир *FastString* . Заметим, однако, что эта функция была экспортирована с опцией *extern "C"* , которая подавляет коррекцию символов. Следовательно, все трансляторы C++ ожидают, что импортируемая библиотека и DLL экспортируют один и тот же идентификатор. Полезным результатом этой методики является то, что вы можете спокойно извлечь класс из DLL, использующей одну среду C++, а обратиться к этому классу из любой другой среды C++. Эта возможность необходима при построении основы для независимых от разработчика компонентов повторного пользования.

Полиморфизм на этапе выполнения

Управление реализациями классов с использованием абстрактных базовых классов как интерфейсов открывает целый мир новых возможностей в терминах того, что может случиться на этапе выполнения. Напомним, что DLL *FastString* экспортирует только один идентификатор – *CreateFastString* . Теперь пользователю легко динамически загрузить DLL, используя по требованию *LoadLibrary* , и разрешить этой единственной точке входа использовать *GetProcAddress* :

```
IFastString *CallCreateFastString(const char *psz)

static IFastString * (*pfn)(const char *) = 0;
if (!pfn)
// init ptr 1st time through
// первое появление ptr
const TCHAR szDll[] = TEXT(«FastString.DLL»);
const char szFn[] = «CreateFastString»;
HINSTANCE h = LoadLibrary(szDll);
if (h) *(FARPROC*)&pfn = GetProcAddress(h, szFn);
return pfn ? pfn(psz) : 0;
```

Эта методика имеет несколько возможных приложений. Одна из причин ее использования – предотвращение ошибок, генерируемых операционной системой при работе на машине, где не установлена реализация объектов. Приложения, использующие дополнительные системные компоненты, такие как WinSock или MAPI, используют похожую технику для запуска приложений на машинах с минимальной конфигурацией. Поскольку клиенту никогда не нужно компоновать импортируемую библиотеку DLL, он не зависит от загрузки DLL и может работать на машинах, на которых DLL вообще не установлена. Другой причиной для использования этой методики может быть медленная

инициализация адресного пространства. Кроме того, DLL не загружается автоматически во время инициализации; и если в действительности реализация объекта не используется, то DLL не загрузится никогда. Другими преимуществами этого способа являются ускорение запуска клиента и сохранение адресного пространства для длительных процессов, которые могут никогда реально не использовать DLL.

Возможно, одним из наиболее интересных применений этой методики является возможность для клиента динамически выбирать между различными реализациями одного и того же интерфейса. Если описание интерфейса *IFastString* дано как общедоступное (publicly available), то ничто не препятствует как исходному конструктору (implementor) *FastString* , так и любым сторонним средствам реализации порождать дополнительные классы реализации от того же самого интерфейса. Подобно исходной реализации класса *FastString* , эти новые реализации будут иметь такое двоичное представление, что будут совместимы на двоичном уровне с исходным классом интерфейса. Все, что должен сделать пользователь, чтобы добиться полностью совместимых («plug-compatible») реализаций, – это определить правильное имя файла для желаемой реализации DLL.

Чтобы понять, как применить эту методику, предположим, что исходная реализация *IFastString* выполняла поиск слева направо. Это прекрасно для языков, анализируемых слева направо (например, английский, французский, немецкий). Для языков, анализируемых справа налево, предпочтительней вторая реализация *IFastString* , осуществляющая поиск справа налево. Эта альтернативная реализация может быть построена как вторая DLL с характерным именем (например, *FastStringRL.DLL*). Пусть обе DLL установлены на машине конечного пользователя, тогда он может выбрать нужный вариант *IFastString* простой загрузкой требуемой DLL на этапе выполнения:

```
IFastString * CallCreateFastString(const char *psz, bool bLeftToRight = true)

static IFastString * (*pfnlr)(const char *) = 0;
static IFastString * (*pfnrl)(const char *) = 0;
IFastString *(**ppfn) (const char *) = &pfnlr;
const TCHAR *pszDll = TEXT(«FastString.DLL»);
if (!bLeftToRight)  pszDll = TEXT(«FastStringRL.DLL»);
ppfn = &pfnrl;
if (!(*ppfn))
// init ptr 1st time through
// первое появление ptr
const char szFn[] = «CreateFastString»;
HINSTANCE h = LoadLibrary(pszDll);
if (h) *(FARPROC*)ppfn = GetProcAddress(h, szFn);
return (*ppfn) ? (*ppfn)(psz) : 0;
```

Когда клиент вызывает функцию без второго параметра,

```
pfs = CallCreateFastString(«Hi Bob!»);
n = pfs->Find(«ob»);
```

то загружается исходная DLL *FastString* , и поиск идет слева направо. Если же клиент указывает, что строка написана на разговорном языке, анализируемом справа налево:

```
pfs = CallCreateFastString(«Hi Bob!», false);
n = pfs->Find(«ob»);
```

то загружается альтернативная версия DLL (*FastStringRL.DLL*), и поиск будет начинаться с крайней правой позиции строки. Главное здесь то, что вызывающие операторы *CallCreateFastString* не заботятся о том, какая из DLL используется для реализации методов объекта. Существенно лишь то, что указатель на совместимый с *IFastString vptr* возвращается функцией и что *vptr* обеспечивает успешное и семантически корректное функционирование. Эта форма полиморфизма на этапе выполнения чрезвычайно полезна при создании системы, динамически скомпонованной из двоичных компонентов.

Расширяемость объекта

Описанные до сих пор методики позволяют клиентам выбирать и динамически загружать двоичные компоненты, что дает возможность изменять с течением времени двоичное представление их реализации без необходимости повторной трансляции клиента. Это само по себе чрезвычайно полезно при построении динамически компонуемых систем. Существует, однако, один аспект объекта, который не может изменяться во времени, – это его интерфейс. Это связано с тем, что пользователь осуществляет трансляцию с определенной сигнатурой класса интерфейса, и любые изменения в описании интерфейса требуют повторной трансляции клиента для учета этих изменений. Хуже того, изменение описания интерфейса полностью нарушает инкапсуляцию объекта (так как его открытый интерфейс изменился) и может испортить программы всех существующих клиентов. Даже самое безобидное изменение, такое как изменение семантики метода с сохранением его сигнатуры, делает бесполезной всю установленную клиентскую базу. Это означает, что интерфейсы являются постоянными двоичными и семантическими контрактами (contracts), которые никогда не должны изменяться. Эта неизменяемость требует стабильной и предсказуемой среды на этапе выполнения.

Несмотря на неизменяемость интерфейсов, часто возникает необходимость добавить дополнительные функциональные возможности, которые не могли быть предусмотрены в период первоначального составления интерфейса. Хотелось бы, например, использовать знание двоичного представления таблицы vtbl и просто добавлять новые методы в конец существующего описания интерфейса. Рассмотрим исходную версию IFastString:

```
class IFastString
public:
virtual void Delete(void) = 0;
virtual int Length(void) = 0;
virtual int Find(const char *psz) = 0;
;
```

Простое изменение класса интерфейса путем объявлений добавочных виртуальных функций *после объявлений существующих методов* имело бы следствием двоичный формат таблицы vtbl, который является надмножеством исходной версии по мере того, как появятся какие-либо новые элементы vtbl после тех, которые соответствуют исходным методам. У реализации объектов, которые транслируются с новым описанием интерфейса, все новые методы будут *добавляться* к исходному размещению vtbl:

```
class IFastString
public:
// faux version 1.0
// фиктивная версия 1.0
virtual void Delete(void) = 0;
virtual int Length(void) = 0;
```

```
virtual int Find(const char *psz) = 0;
// faux version 2.0
// фиктивная версия 2.0
virtual int FindN(const char *psz, int n) = 0;
;
```

Это решение почти работает. Те клиенты, у которых оттранслирована исходная версия интерфейса, остаются в счастливом неведении относительно всех составляющих таблицы vtbl, кроме первых трех. Когда старые клиенты получают обновленные объекты, имеющие в vtbl вход для FindN, они продолжают нормально работать. Проблема возникает, когда новым клиентам, ожидающим, что IFastString имеет четыре метода, случится столкнуться с устаревшими объектами, где метод FindN не реализуется. Когда клиент вызовет FindN на объект, транслированный с исходным описанием интерфейса, результаты будут вполне определенными. Программа прервет работу.

В этой методике проблема заключается в том, что она нарушает инкапсуляцию объекта, изменяя открытый интерфейс. Подобно тому, как изменение открытого интерфейса в классе C++ может вызвать ошибки на этапе трансляции, когда происходит перестройка клиентского кода, так и изменение двоичного описания интерфейса вызовет ошибки на этапе выполнения, когда клиентская программа перезапущена. Это означает, что интерфейсы должны быть неизменяемыми с момента первой редакции. Решение этой проблемы заключается в том, чтобы разрешить классу реализации выставлять более чем один интерфейс. Этого можно достигнуть, если предусмотреть, что один интерфейс порождается от другого, связанного с ним интерфейса. А можно сделать так, чтобы класс реализации наследовал от нескольких несвязанных классов интерфейса. В любом случае клиент мог бы использовать имеющуюся в C++ возможность определения типа на этапе выполнения – идентификацию Runtime Type Identification – RTTI, чтобы динамически опросить объект и убедиться в том, что его требуемая функциональность действительно поддерживается уже работающим объектом.

Рассмотрим простой случай интерфейса, расширяющего другой интерфейс. Чтобы добавить в IFastString операцию FindN, позволяющую находить *n* –е вхождение подстроки, необходимо породить второй интерфейс от IFastString и добавить в него новое описание метода:

```
class IFastString2 : public IFastString
public: // real version 2.0
// настоящая версия 2.0
virtual int FindN(const char *psz, int n) = 0;
;
```

Клиенты могут с уверенностью динамически опрашивать объект с помощью оператора C++ dynamic_cast, чтобы определить, является ли он совместимым с IFastString2

```
int Find10thBob(IFastString *pfs)
IFastString2 *pfs2 = dynamic_cast<IFastString2*>(pfs);
if(pfs2)
// the object derives from IFastString2
// объект порожден от IFastString2
return pfs2->FindN(«Bob», 10);
else
// object doesn't derive from IFastString2
// объект не порожден от IFastString2
error(«Cannot find 10th occurrence of Bob»);
```

```
return -1;
```

Если объект порожден от расширенного интерфейса, то оператор dynamic_cast возвращает указатель на вариант объекта, совместимый с IFastString2, и клиент может вызвать расширенный метод объекта. Если же объект не порожден от расширенного интерфейса, то оператор dynamic_cast возвратит пустой (null) указатель. В этом случае клиент может или выбрать другой способ реализации, зарегистрировав сообщение об ошибке, или молча продолжить без расширенной операции. Эта способность назначенного клиентом постепенного сокращения возможностей очень важна при создании гибких динамических систем, которые могут обеспечить со временем расширенные функциональные возможности.

Иногда требуется раскрыть еще один аспект функциональности объекта, тогда разворачивается еще более интересный сценарий. Обсудим, что следует предпринять, чтобы добавить постоянства, или персистентности (persistence), классу реализации IFastString. Хотя, вероятно, можно добавить методы Load и Save к расширенной версии IFastString, другие типы объектов, не совместимые с IFastString, могут тоже быть постоянными. Простое создание нового интерфейса, который расширяет IFastString:

```
class IPersistentObject : public IFastString

public: virtual bool Load(const char *pszFileName) = 0;
virtual bool Save(const char *pszFileName) = 0;
;
```

требует, чтобы все постоянные объекты поддерживали также операции Length и Find. Для некоторого, весьма малого подмножества объектов это могло бы иметь смысл. Однако для того, чтобы сделать интерфейс IPersistentObject возможно более общим, он должен быть своим собственным интерфейсом, а не порождаться от IFastString:

```
class IPersistentObject

public: virtual void Delete(void) = 0;
virtual bool Load(const char *pszFileName) = 0;
virtual bool Save(const char *pszFileName) = 0;
;
```

Это не мешает реализации FastString стать постоянной; это просто означает, что постоянная версия FastString должна поддерживать оба интерфейса: и IFastString, и IPersistentObject:

```
class FastString : public IFastString, public IPersistentObject

int m_cch;
// count of characters
// счетчик символов
char *m_psz;
public: FastString(const char *psz);
~FastString(void);
// Common methods
// Общие методы
void Delete(void);
```

```
// deletes this instance
// уничтожает этот экземпляр

// IFastString methods
// методы IFastString
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение

// IPersistentObject methods
// методы IPersistentObject
bool Load(const char *pszFileName);
bool Save(const char *pszFileName);
;
```

Чтобы записать FastString на диск, пользователю достаточно с помощью RTTI связать указатель с интерфейсом IPerststentObject, который выставляется объектом:

```
bool SaveString(IFastString *pfs, const char *pszFN)

bool bResult = false;
IPersistentObject *ppo = dynamic_cast<IPersistentObject*>(pfs);
if (ppo) bResult = ppo->Save(pszFN);
return bResult;
```

Эта методика работает, поскольку транслятор имеет достаточно информации о представлении и иерархии типов класса реализации, чтобы динамически проверить объект для выяснения того, действительно ли он порожден от IPersistentObject. Но здесь есть одна проблема.

RTTI – особенность, сильно зависящая от транслятора. В свою очередь, DWP передает синтаксис и семантику RTTI, но каждая реализация RTTI разработчиком транслятора уникальна и запатентована. Это обстоятельство серьезно подрывает независимость от транслятора, которая была достигнута путем использования абстрактных базовых классов как интерфейсов. Это является неприемлемым для архитектуры компонентов, не зависимой от разработчиков. Удачным решением было бы упорядочение семантики dynamic_cast без использования свойств языка, зависящих от транслятора. Явное выставление хорошо известного метода из каждого интерфейса, представляющего семантический эквивалент dynamic_cast, позволяет достичь желаемого эффекта, не требуя, чтобы все объекты использовали тот же самый транслятор C++:

```
class IPersistentObject

public: virtual void *Dynamic_Cast(const char *pszType) = 0;
virtual void Delete(void) = 0;
virtual bool Load(const char *pszFileName) = 0;
virtual bool Save(const char *pszFileName) = 0;
;
class IFastString
```

```
public: virtual void *Dynamic_Cast(const char *pszType) = 0;
virtual void Delete(void) = 0;
virtual int Length(void) = 0;
virtual int Find(const char *psz) = 0;
;
```

Так как всем интерфейсам необходимо выставить этот метод вдобавок к уже имеющемуся методу Delete, имеет большой смысл включить общее подмножество методов в базовый интерфейс, из которого могли бы порождаться все последующие интерфейсы:

class IExtensibleObject public: virtual void *Dynamic_Cast(const char* pszType) = 0; virtual void Delete(void) = 0; ; class IPersistentObject : public IExtensibleObject public: virtual bool Load(const char *pszFileName) = 0; virtual bool Save(const char *pszFileName) = 0; ; class IFastString : public IExtensibleObject public: virtual int Length(void) = 0; virtual int Find(const char *psz) = 0; ;

Имея такую иерархию типов, пользователь может динамически запросить объект о данном интерфейсе с помощью следующей не зависящей от транслятора конструкции:

bool SaveString(IFastString *pfs, const char *pszFN) boot bResult = false; IPersistentObject *ppo = (IPersistentObject) pfs->Dynamic_Cast(«IPers1stentObject»); if (ppo) bResult = ppo->Save(pszFN); return bResult;

В только что приведенном примере клиентского использования присутствуют требуемая семантика и механизм для определения типа, но каждый класс реализации должен выполнять это функциональное назначение самолично:

class FastString : public IFastString, public IPersistentObject

```
int m_cch;
// count of characters
// счетчик символов
char *m_psz;
public:
FastString(const char *psz);
~FastString(void);
// IExtensibleObject methods
// методы IExtensibleObject
void *Dynamic_Cast(const char *pszType);
void Delete(void);
// deletes this instance
// удаляет этот экземпляр
// IFastString methods
// методы IFastString
int Length(void) const;
// returns # of characters
// возвращает число символов
int Find(const char *psz) const;
// returns offset
// возвращает смещение
// IPersistentObject methods
```

```
// методы IPersistentObject
bool Load(const char *pszFileName);
bool Save(const char *pszFileName);
;
```

Реализации Dynamic_Cast необходимо имитировать действия RTTI путем управления иерархией типов объекта. Рисунок 1.8 иллюстрирует иерархию типов для только что показанного класса FastString. Поскольку класс реализации порождается из каждого интерфейса, который он выставляет, реализация Dynamic_Cast в FastString может просто использовать явные статические приведения типа (explicit static casts), чтобы ограничить область действия указателя this, основанного на подтипе, который запрашивается клиентом:

```
void *FastString::Dynam1c_Cast(const char *pszType)

if (strcmp(pszType, «IFastString») == 0) return static_cast<IFastString*>(this);
else if (strcmp(pszType, «IPersistentObject») == 0) return
static_cast<IPersistentObject*>(this);
else if (strcmp(pszType, «IExtensibleObject») == 0) return static_cast<IFastString*>(this);
else return 0;
// request for unsupported interface
// запрос на неподдерживаемый интерфейс
```

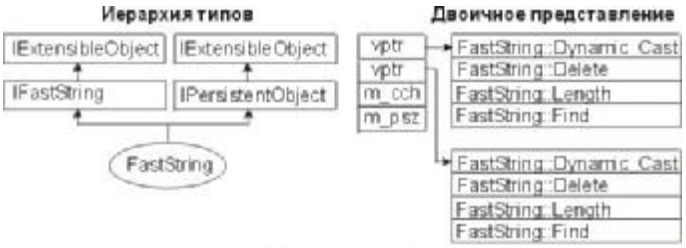

Рис. 1.8. Иерархия типов FastString

Так как объект порождается от типа, используемого в этом преобразовании, оттранслированные версии операторов преобразования просто добавляют определенное смещение к указателю объекта this, чтобы найти начало представления базового класса.

Отметим, что после запроса на общий базовый интерфейс IExtensibleObject реализация статически преобразуется в IFastString. Это происходит потому, что интуитивная версия (intuitive version) оператора

```
return static_cast<IExtensibleObject*>(this);
```

неоднозначна, так как и IFastString, и IPersistentObject порождены от IExtensibleObject. Если бы IExtensibleObject был виртуальным базовым классом как для IFastString, так и для IPersistentObject, то данное преобразование не было бы неоднозначным и оператор бы оттранслировался. Тем не менее, применение виртуальных базовых классов добавляет на этапе выполнения ненужную сложность в результирующий объект и к тому же вносит зависимость от транслятора. Дело в том, что виртуальные базовые классы являются всего лишь особенностями языка C++, которые имеют несколько специфических реализации.

Управление ресурсами

Еще одна проблема поддержки нескольких интерфейсов из одного объекта становится яснее, если исследовать схему использования клиентом метода *DynamicCast* . Рассмотрим

следующую клиентскую программу:

```
void f(void)

IFastString *pfs = 0;
IPersistentObject *ppo = 0;
pfs = CreateFastString(«Feed BOB»);
if (pfs)
ppo = (IPersistentObject *) pfs->Dynamic
Cast(«IPersistentObject»);
if ( !ppo) pfs->Delete();
else  ppo->Save(«C:\autoexec.bat»);
ppo->Delete();
```

Хотя вначале объект был связан через свой интерфейс *IFastString* , клиентский код вызывает метод *Delete* через интерфейс *IPersistentObject* . С использованием свойства C++ о множественном наследовании это вполне допустимо, так как все таблицы *vtbl* , порожденные классом *IExtensibleObject* , укажут на единственную реализацию метода *Delete* . Теперь, однако, пользователь должен хранить информацию о том, какие указатели связаны с какими объектами, и вызывать *Delete* только один раз на объект. В случае простого кода, приведенного выше, это не слишком тяжелое бремя. Для более сложных клиентских кодов управление этими связями становится делом весьма сложным и чреватым ошибками. Одним из способов упрощения задачи пользователя является возложение ответственности за управление жизненным циклом объекта на реализацию. Кроме того, разрешение клиенту явно удалять объект вскрывает еще одну деталь реализации: тот факт, что объект находится в динамически распределяемой памяти (в «куче», on the heap).

Простейшее решение этой проблемы – ввести в каждый объект счетчик ссылок, который увеличивается, когда указатель интерфейса дублируется, и уменьшается, когда указатель интерфейса уничтожается. Это предполагает изменение определения *IExtensibleObject* с

```
class IExtensibleObject

public:
virtual void *DynamicCast (const char* pszType) =0;
virtual void Delete(void) = 0;
;
```

на

```
class IExtensibleObject

public:
virtual void *DynamicCast(const char* pszType) = 0;
virtual void DuplicatePointer(void) = 0;
virtual void DestroyPointer(void) = 0;
;
```

Разместив эти методы, все пользователи *IExtensibleObject* должны теперь придерживаться следующих двух соображений:

1) Когда указатель интерфейса дублируется, требуется вызов *DuplicatePointer* .

2) Когда указатель интерфейса более не используется, следует вызвать *DestroyPointer* .

Эти методы могут быть реализованы в каждом объекте: нужно просто фиксировать количество действующих указателей и уничтожать объект, когда невыполненных указателей не осталось:

```
class FastString : public IFastString,
public IPersistentObject

int mcPtrs;
// count of outstanding ptrs
// счетчик невыполненных указателей
public:
// initialize pointer count to zero
// сбросить счетчик указателя в нуль
FastString(const char *psz) : mcPtrs(0)
void DuplicatePointer(void)

// note duplication of pointer
// отметить дублирование указателя
++mcPtrs;

void DestroyPointer(void)

// destroy object when last pointer destroyed
// уничтожить объект, когда уничтожен последний указатель
if (-mcPtrs == 0) delete this;

: : :
;
```

Этот совершенно стандартный код мог бы просто быть включен в базовый класс или в макрос С-препроцессора, чтобы его могли использовать все реализации.

Чтобы поддерживать эти методы, все программы, которые манипулируют или управляют указателями интерфейса, должны придерживаться двух простых правил *DuplicatePointer/DestroyPointer* . Для реализации *FastString* это означает модификацию двух функций. Функция *CreateFastString* берет начальный указатель, возвращаемый новым оператором C++, и копирует его в стек для возврата клиенту. Следовательно, необходим вызов *DuplicatePointer* :

```
IFastString* CreateFastString(const char *psz)

IFastString *pfsResult = new FastString(psz);
if (pfsResult) pfsResult->DuplicatePointer();
return pfsResult;
```

Реализация копирует указатель и в другом месте – в методе Dynamic_Cast:

```
void *FastString::Dynamic_Cast(const char *pszType)

void *pvResult = 0;
if (strcmp(pszType, «IFastString») == 0) pvResult = static_cast<IFastString*>(this);
```

```
    else if (strcmp(pszType, «IPersistentObject») == 0) pvResult =
static_cast<IPersistentObject*>(this);
    else if (strcmp(pszType, «IExtensibleObject») == 0) pvResult =
static_cast<IFastString*>(this);
    else return 0;
    // request for unsupported interface
    // запрос на неподдерживаемый интерфейс
    // pvResult now contains a duplicated pointer, so
    // we must call DuplicatePointer prior to returning
    // теперь pvResult содержит скопированный указатель,
    // поэтому нужно перед возвратом вызвать DuplicatePointer
    ((IExtensibleObject*)pvResult)->DuplicatePo1nter();
    return pvResult;
```

С этими двумя усовершенствованиями соответствующий код пользователя становится значительно более однородным и прозрачным:

```
void f(void)

IFastString *pfs = 0;
IPersistentObject *ppo = 0;
pfs = CreateFastString(«Feed BOB»);
if (pts)
ppo = (IPersistentObject *) pfs->DynamicCast(«IPersistentObject»);
if (ppo)  ppo->Save(«C:\autoexec.bat»);
ppo->DestroyPointer();
pfs->DestroyPointer();
```

Поскольку каждый указатель теперь трактуется как автономный объект с точки зрения времени жизни, клиенту можно не интересоваться тем, какой указатель соответствует какому объекту. Вместо этого клиент просто придерживается двух простых правил и предоставляет объектам самим управлять своим временем жизни. При желании способ вызова *DuplicatePointer* и *DestroyPointer* можно легко скрыть за интеллектуальным указателем (smart pointer) C++.

Использование этой схемы вычисления ссылок позволяет объекту весьма единообразно выставлять множественные интерфейсы. Возможность выставления нескольких интерфейсов из одного класса реализации позволяет типу данных участвовать в различных контекстах. Например, новая постоянная подсистема могла бы определить собственный интерфейс для управления автозагрузкой и автозаписью объектов на некоторый специализированный носитель. Класс *FastString* мог бы добавить поддержку этих возможностей простым наследованием от постоянного интерфейса этой подсистемы. Добавление этой поддержки никак не повлияет на уже установленные базы клиентов, которые, может быть, используют прежний постоянный интерфейс для записи и загрузки строки на диск. Механизм согласования интерфейсов на этапе выполнения может служить краеугольным камнем для построения динамической системы из компонентов, которые могут изменяться со временем.

Где мы находимся?

Мы начали эту главу с простого класса C++ и рассмотрели проблемы, связанные с объявлением этого класса как двоичного компонента повторного использования. Первым

шагом было употребление этого класса в качестве библиотеки *Dynamic Link Library* (DLL) для отделения физической упаковки этого класса от упаковок его клиентов. Затем мы использовали понятие интерфейсов и реализации для инкапсуляции элементов реализации типов данных за двоичной защитой, что позволило изменять двоичные представления объектов без необходимости перетрансляции клиентами. Затем, используя для определения интерфейсов подход абстрактного базового класса, эта защита приобрела форму указателя *vptr* и таблицы *vtbl* . Далее мы исследовали приемы для динамического выбора различных полиморфных реализаций данного интерфейса на этапе выполнения с использованием *LoadLibrary* и *GetProcAddress* . Наконец, мы использовали RTTI-подобную структуру для динамического опроса объекта с целью определить, действительно ли он использует нужный интерфейс. Эта структура предоставила нам методику расширения существующих версий интерфейса, а также возможность выставления нескольких несвязанных интерфейсов из одного объекта.

Короче, мы только что создали модель компонентных объектов (*Component Object Model – COM*).

Глава 2. Интерфейсы

```
void *pv = malloc(sizeof(int));
int *pi = (int*)pv;
(*pi)++;
free(pv);
```
Аноним,1982

В предыдущей главе было показано несколько приемов программирования на C++, позволяющих разрабатывать двоичные компоненты повторного использования, которые со временем могут быть модернизированы. По своему смыслу эти приемы идентичны тем, которые используются моделью COM. Незначительные различия между методиками предыдущей главы и теми, которые используются COM, в большинстве случаев заключаются в деталях и почти всегда достаточно обоснованы. Вообще-то предыдущая глава прослеживала историю модели COM, которая прежде всего и в основном есть отделение интерфейса от реализации.

Снова об интерфейсах и реализациях

Снова об интерфейсах и реализациях

Цель отделения интерфейса от реализации заключалась в сокрытии от клиента всех деталей внутренней работы объекта. Этот фундаментальный принцип предусматривал уровень косвенности, или изоляции (*level of indirection*), который позволял изменяться количеству или порядку элементов данных в реализации класса без перекомпиляции клиента. Кроме того, этот принцип позволял клиентам обнаруживать расширенную функциональность путем опроса объекта на этапе выполнения. И, наконец, этот принцип позволяет сделать библиотеку DLL независимой от транслятора C++, который используется клиентом.

Хотя этот последний аспект и полезен, он далеко не достаточен для обеспечения универсальной основы для двоичных компонентов. Важно отметить, что хотя клиенты могут использовать любой выбранный ими транслятор C++, в конечном счете это будет всего лишь транслятор C++. Приемы, описанные в предыдущей главе, обеспечивают независимость от

транслятора. В конце концов, главное, что необходимо для создания действительно универсальной основы для двоичных компонентов, – это независимость от языка. А чтобы достичь независимости от языка, принцип отделения интерфейса от реализации должен быть применен еще раз.

Рассмотрим определения интерфейса, использованные в предыдущей главе. Каждое определение интерфейса принимало форму определения абстрактного базового класса C++ в заголовочном файле C++. Тот факт, что определение интерфейса постоянно находится в файле, читаемом только на одном языке, вскрывает один остаточный признак реализации этого объекта – язык, на котором он был написан. Но, в конечном счете, объект должен быть доступен для любого языка, а не только для того, который выбрал разработчик объекта. Предусматривая только совместимое с C++ определение интерфейса, разработчик объекта тем самым вынуждает всех использующих этот объект также работать на C++.

Хотя C++ – чрезвычайно полезный язык программирования, существует множество областей программирования, где больше подходят другие языки. Но точно так же, как проблемы совместимости при компоновке можно решить путем обеспечения всех существующих компиляторов файлами определения модуля, возможно и перевести определение интерфейса с C++ на любые другие языки программирования. А так как двоичная сигнатура интерфейса есть просто сочетание *vptr/vtbl* , этот перевод может быть сделан для большой группы языков.

Проделывание этих языковых преобразований данных для всех известных интерфейсов потребовало бы огромного количества работы, а главное – невозможно успевать делать это для бурного потока языков программирования, которые индустрия программного обеспечения не устает изобретать чуть ли не каждую декаду. Идеально было бы написать сервисную программу, которая переводила бы определения класса C++ в некую абстрактную промежуточную форму. Из этой промежуточной формы такая программа могла бы преобразовывать данные для любого языка программирования, имеющего соответствующий выходной генератор (back-end generator). По мере того как новые языки приобретают значимость, могли бы добавляться новые выходные генераторы, и все ранее определенные интерфейсы смогли бы тотчас использоваться в совершенно новом контексте.

К сожалению, язык программирования C++ полон неоднозначностей, что делает его малопригодным для преобразования данных на все мыслимые языки. Многие из этих неоднозначностей приводят к неопределенным соотношениям между указателями, памятью и массивами. Это не является проблемой, когда оба объекта: вызывающий (*caller*) и вызываемый (*callee*) – скомпилированы на C или на C++, но они не могут быть точно переведены на другие языки без дополнительной квалификации. Поэтому, чтобы устранить зависимость определения интерфейса от языка, используемого в какой-либо конкретной реализации, необходимо для определений *интерфейсов* использовать один язык, а для определений *реализации – другой.* Если все участники договорятся о едином языке для определений интерфейсов, то станет возможным определить интерфейс однажды и получать по мере необходимости новые представления реализации на специфических языках. COM предусматривает язык, который основан на хорошо известном синтаксисе C, но добавляет возможность при переводе на другие языки корректно устранить неоднозначность любых особенностей языка C. Этот язык называется языком описаний интерфейса (Interface Definition Language – *IDL*).

IDL

COM IDL базируется на языке определения интерфейсов основного открытого математического обеспечения удаленного вызова процедур в распределенной вычислительной среде – Open Software Foundation Distributed Computing Environment Remote Procedure Call (OSF DCE RPC). DCE IDL позволяет описывать удаленные вызовы процедур

не зависящим от языка способом. Это дает возможность компилятору IDL генерировать код для работы в сети, который прозрачным образом (transparently), то есть незаметно для пользователя, переносит описанные операции на всевозможные сетевые средства сообщения. COM IDL просто добавляет некоторые расширения, специфические для COM, в DCE IDL для поддержки объектно-ориентированных понятий COM (например, наследование, полиморфизм). Не случайно, что когда обращение к объектам COM осуществляется через границу контекста выполнения[6] или через границы между машинами, все связи клиент-объект используют *MS-RPC* (реализация DCE RPC, являющаяся частью Windows NT и Windows 95) как основное средство сообщения.

Win32 SDK включает в себя компилятор *MIDL.EXE* , который анализирует файлы *COM IDL* и генерирует несколько искусственных объектов – артефактов (artifacts). Как показано на рис. 2.1, MIDL генерирует совместимые с C/C++ заголовочные файлы, которые содержат определения абстрактного базового класса, соответствующие интерфейсам, описанным в исходном IDL-файле.

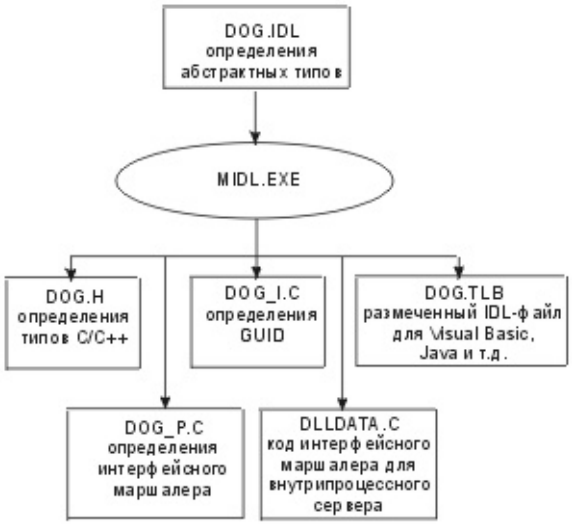

Рис. 2.1. Использование MIDL

Эти заголовочные файлы также содержат совместимые с C, основанные на структурах определения (structure-based definitions), которые позволяют C-программам обращаться к интерфейсам, описанным на IDL, или обеспечивать их выполнение. То, что MIDL автоматически генерирует C/C++-заголовочный файл, означает, что ни один из COM-интерфейсов не нужно определять на C++ вручную. Исход определений из одной точки исключает возникновение множества несовместимых версий определений интерфейсов, которые со временем могут вызвать асинхронность. MIDL также генерирует исходный код, который позволяет использовать интерфейсы в различных потоках, процессах и машинах. Этот код будет обсуждаться в главе 5. И наконец, MIDL может генерировать двоичный файл, который позволяет другим средам, принимающим COM, отображать интерфейсы, определенные в исходном IDL-файле, на другие языки. Этот двоичный файл называется библиотекой типа (*type library*) и содержит разобранный файл IDL в наиболее эффективной для анализа форме. Библиотеки типа обычно распространяются как часть исполняемого файла реализации и позволяют таким языкам, как Visual Basic, Java, Object Pascal использовать интерфейсы, которые выставляются этой реализацией.

Чтобы понять IDL, необходимо рассмотреть логический и физический аспекты

---

[6] 1 Термин *контекст выполнения* (execution context) используется в определении COM, чтобы описать все, что было впоследствии переименовано в *апартамент* (apartment). Апартамент – не поток и не процесс; однако он имеет общие для этих обоих понятии признаки. Подробно понятие апартамента описано в главе 5.

интерфейса. Обсуждение методов интерфейса и выполняемых ими операций относятся к логическому аспекту интерфейса. Обсуждение памяти, стекового фрейма, сетевых пакетов и других динамических явлений обычно относятся к физическому аспекту интерфейса. Некоторые физические аспекты интерфейса могут непосредственно наследовать логическому описанию (например, расположение таблицы *vtbl* , порядок параметров в стеке). Другие физические аспекты (например, границы массивов, сетевые представления сложных типов данных) требуют дополнительной квалификации.

IDL позволяет разработчикам интерфейса работать непосредственно в сфере логики, используя синтаксис C. Но в то же время IDL требует от разработчиков точно определять все те аспекты интерфейса, которые не могут быть воспроизведены непосредственно по их логическому описанию на C, с помощью использования аннотаций, формально называемых атрибутами. Атрибуты IDL легко распознать в основном тексте IDL: разделенные запятыми, они заключены в скобки. Атрибуты всегда предшествуют описанию объекта, к которому они относятся. Например, в следующем IDL– фрагменте

```
[
v1enum , helpstring(«This is a color!»)
]
enum COLOR  RED, GREEN, BLUE ;
```

атрибут v1_enum относится к описанию перечисления (enumeration) COLOR. Этот атрибут информирует компилятор IDL о том, что представление COLOR при передаче значения через сеть должно иметь 32 бита, а не 16, как принято по умолчанию. Атрибут helpstring также относится к COLOR и добавляет строку «This is a color!» («Это – цвет!») в создаваемую библиотеку типа как описание этого перечисления. Если игнорировать атрибуты в IDL-файле, то его синтаксис такой же, как в C. IDL поддерживает структуры, объединения, массивы, перечисления, а также определения типа (typedef) – с синтаксисом, идентичным их аналогам в C.

Определяя методы COM в IDL, необходимо четко указать, кто – вызывающий или вызываемый объект – будет записывать или читать каждый параметр метода. Это выполняется с помощью атрибутов параметра *[in]* и *[out]* :

```
void Method1([in] long arg1, [out] long *parg2, [in, out] long *parg3);
```

Для этого фрагмента IDL предполагается, что вызывающий объект передаст значение в объект *arg1* и по адресу, содержащемуся в указателе *parg3* . По завершении возвращаемые значения будут получены вызывающим объектом по адресам, указанным в *parg2* и *parg3* . Это означает, что для последовательности вызовов:

```
long arg2 = 20, arg3 = 30;
p->Method1(10, &arg2, &arg3);
```

объект не может полагаться на получение передаваемого значения *20* через *parg2* . Если объект запускается в том же контексте выполнения, что и вызывающий объект, и оба участника вызова реализованы на C++, то **parg2* действительно будет иметь на входе метода значение *20* . Однако если объект вызывается из другого контекста выполнения или один из участников вызова реализован на языке, который сводит на нет оптимизацию начальных значений чисто выходных (*out-only*) параметров, то инициализация параметра вызывающим объектом будет утеряна.

Методы и их результаты

Результаты методов – это одна из сторон COM, где логический и физический миры расходятся. В сущности, все методы COM физически возвращают номер ошибки с типом *HRESULT* . Использование одного типа возвращаемого результата позволяет удаленной COM-архитектуре перегружать результат выполнения метода, а также сообщать об ошибках соединения, просто зарезервировав ряд величин для RPC-ошибок. Величины *HRESULT*

представляют собой 32-битные целые числа, которые передают в вызывающий контекст выполнения информацию о типе ошибок, которые могут произойти (например, ошибки сети, сбои сервера). Во многих языках, поддерживающих COM (например, Visual Basic, Java), ***HRESULT*** –значения перехватываются контекстом выполнения или виртуальной машиной и преобразуются в программные исключения (programmatic exceptions).

Рис. 2.2. HRESULT

Как показано на рис. 2.2, HRESULT-значения состоят из трех битовых полей: бита серьезности ошибки (severity bit), кода устройства и информационного кода. Бит серьезности ошибки показывает, успешно выполнена операция или нет, код устройства индицирует, к какой технологии относится ***HRESULT*** , а информационный код представляет собой точный результат в рамках заданной технологии и серьезности. Заголовки SDK (software development kit – набор инструментальных средств разработки программного обеспечения) определяют два макроса, облегчающие работу с HRESULT:

#define SUCCEEDED(hr) (long(hr) >= 0) #def1ne FAILED(hr) (long(hr) < 0)

Эти два макроса используют тот факт, что при трактовке ***HRESULT*** как целого числа со знаком бит серьезности ошибки он является также знаковым битом.

Заголовки SDK содержат определения всех стандартных ***HRESULT*** . Эти ***HRESULT*** имеют символические имена, соответствующие трем компонентам ***HRESULT*** , и используются в следующем формате:

<facility>_<*severity*>_ <information>

Например, HRESULT с именем STG_S_CONVERTED показывает, что кодом устройства является FACILITY_STORAGE. Это означает, что результат относится к структурированному хранилищу (Structured Storage) или к персистентности (Persistence). Код серьезности ошибки – SEVERITY_SUCCESS. Это означает, что вызов смог успешно выполнить операцию. Третья составляющая – CONVERTED – означает, что в данном случае было произведено преобразование базового файла для поддержки структурированного хранилища. HRESULT-значения, являющиеся универсальными и не привязанными к определенной технологии, используют FACILITY_NULL, и их символическое имя не содержит префикса кода устройства. Вот некоторые стандартные имена HRESULT-значений с кодом FACILITY_NULL:

S_OK – успешная нормальная операция
S_FALSE – используется для возвращения логического false в случае успеха
E_FAIL – общий сбой E_NOTIMPL – метод не реализован
E_UNEXPECTED – метод вызван в неподходящее время

FACILITY_ITF используется в специфически интерфейсных HRESULT-значениях и является в то же время единственным допустимым кодом устройства для HRESULT, определяемых пользователем. При этом значения FACILITY_ITF должны быть уникальными в контексте каждого отдельного интерфейса. Стандартные заголовки определяют макрос MAKE_HRESULT для определения пользовательского HRESULT из трех необходимых полей:

const HRESULT CALC_E_IAMHOSED = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, 0x200 + 15);

Для пользовательских *HRESULT* принято соглашение, что значения информационного кода должны превышать *0x200* , чтобы избежать повторного использования значений, уже задействованных в системных *HRESULT* -значениях. Хотя это не опасно, таким образом предотвращается повторное использование значений, уже имеющих смысл для стандартных интерфейсов. Например, большинство HRESULT имеют текстовые описания для пользователя, которые можно получить на этапе выполнения с помощью функции API *FormatMessage* . Выбор *HRESULT* , не пересекающихся со значениями, определенными в системе, служит гарантией того, что неверные сообщения об ошибках не будут получены.

Чтобы позволить методам возвращать логический результат, не имеющий отношения к их физическому *HRESULT* -значению, язык COM IDL поддерживает атрибут параметров *retval* . Атрибут *retval* показывает, что соответствующий параметр физического метода в действительности является логическим результатом операции и, если контекст это позволяет, должен быть представлен как результат операции. Рассмотрим IDL-описание следующего метода:

HRESULT Method2([in] short arg1, [out, retval] short *parg2);

на языке Java это соответствует:

public short Method2(short arg1);

в то время как Visual Basic дает такое описание метода:

Function Method2(arg1 as Integer) As Integer

Поскольку C++ не использует поддержку контекста выполнения для обращения к COM-интерфейсам, представление этого метода в Microsoft C++ имеет вид:

virtual HRESULT *stdcall Method2(short arg1* , short *parg2) = 0;

Это значит, что следующий клиентский код на языке C++:

```
short sum = 10;
short s;
HRESULT hr = pItf->Method2(20, &s);
if (FAILED(hr)) throw hr;
sum += s;
```

примерно эквивалентен такому Java-коду:

short sum == 10; short s = Itf.Method2(20); sum += s;

Если *HRESULT* , возвращенный методом, сообщает об аварийном результате, то Java Virtual Machine преобразует *HRESULT* в исключение Java. Во фрагменте кода на языке C++ необходимо проверить вручную *HRESULT* , возвращенный этим методом, и соответствующим образом обработать этот аварийный результат.

Интерфейсы и IDL

Определения методов в IDL являются просто аннотированными аналогами C-функций.

Определения интерфейсов в IDL требуют расширения по сравнению с C, так как C не имеет встроенной поддержки этого понятия. Определение интерфейса в IDL начинается с ключевого слова ***interface*** . Это определение состоит их четырех частей: имя интерфейса, базовое имя интерфейса, тело интерфейса и атрибуты интерфейса. Тело интерфейса представляет собой просто набор определений методов и операторов определения типов:

```
[ attribute1, attribute2, …]
interface IThisInterface : IBaseInterface

typedef1;
typedef2;
:
:
method1;
method2;
```

Каждый интерфейс COM должен иметь как минимум два атрибута IDL. Атрибут ***[object]*** служит признаком того, что данный интерфейс является COM-, а не DCE-интерфейсом. Второй обязательный атрибут указывает на физическое имя интерфейса (в предшествующем IDL-фрагменте ***IThisInterface*** является логическим именем интерфейса).

Чтобы понять, почему COM-интерфейсы требуют физическое имя, отличное от логического имени интерфейса, рассмотрим следующую ситуацию. Два разработчика независимо друг от друга решили создать интерфейс, моделирующий ручной калькулятор. Два их определения интерфейса будут, вероятно, похожими, будучи заданными в общей проблемной области, но скорее всего фактический порядок определений методов и, возможно, сигнатур методов могут в чем-то различаться. Несмотря на это, оба разработчика, вероятно, выберут одно и то же логическое имя: ***ICalculator*** .

Клиентская программа на машине какого-нибудь конечного пользователя может реализовать определение интерфейса от первого разработчика, а запустить объект, созданный вторым. Поскольку оба интерфейса имеют одно и то же логическое имя, то если клиент запросит объект для поддержки ***ICalculator*** , просто использовав строку «ICalculator», объект ответит на запрос возвратом ненулевого указателя интерфейса. Однако представление клиента о том, на что похож ***ICalculator*** , вступит в конфликт с тем, какое представление о нем имеет этот объект, и результирующий указатель будет не тем, которого ожидает клиент. Ведь эти два интерфейса могут быть совершенно разными, несмотря на то, что оба используют одно и то же логическое имя.

Чтобы исключить коллизию имен, всем COM-интерфейсам на этапе проектирования назначается уникальное двоичное имя, которое является физическим именем интерфейса. Эти физические имена называются глобально уникальными идентификаторами (***Globally Unique Identifiers – GUIDs***), что рифмуется со словом ***squids*** [7]. GUID используются в COM повсюду для именования статических сущностей, таких как интерфейсы или реализации. GUID являются чрезвычайно большими 128-битными числами, что гарантирует их уникальность как во времени, так и в пространстве. GUID в COM основаны на универсальных уникальных идентификаторах (***Universally Unique Identifiers – UUIDs***), используемых в DCE RPC. При использовании GUID для именования COM-интерфейсов их часто называют идентификаторами интерфейса (***Interface IDs – IIDs***). Реализации в COM также именуются с помощью GUID, и в этом случае GUID называются идентификаторами класса (***Class IDs – CLSIDs***). Будучи представленными в текстовой форме, GUID всегда имеют следующий канонический вид: BDA4A270-A1BA-11d0-8C2C-0080C73925BA

---

[7] *1* Точное произношение слова GUID является предметом горячих споров между разработчиками COM. Хотя спецификация COM формулирует, что GUID рифмуется с ***fluid (подвижный)*** , а не ***squid (кальмар)*** , автор уверен, что она ошибается, ссылаясь как на прецедент на слово ***languid (медлительный)***

Эти 32 шестнадцатеричные цифры представляют 128-битное значение GUID. Именование интерфейсов и реализации с помощью GUID важно для предотвращения коллизий между разными компонентами.

Для создания нового GUID в COM имеется API-функция, которая использует децентрализованный алгоритм уникальности для генерирования нового 128-битного числа, которое никогда больше не встретится в природе:

HRESULT CoCreateGuid(GUID *pguid);

Алгоритм, задействованный в функции *CoCreateGuid,* использует локальный сетевой интерфейсный адрес машины, текущее машинное время и два постоянных счетчика для компенсации точности часов и нестандартных изменении в них (таких, как переход на летнее время или ручная коррекция системных часов). Если данная машина не имеет сетевого интерфейса, то синтезируется статистически уникальная величина и *CoCreateGuid* возвращает особого вида *HRESULT* , показывающий, что данная величина является глобально уникальной только статистически и может считаться таковой только при использовании на локальной машине. Хотя прямой вызов функции *CoCreateGuid* иногда полезен, большинство разработчиков вызывают ее в неявной форме, применяя из SDK программу *GUIDGEN.EXE* . На рис. 2.3 показана работа *GUIDGEN* . *GUIDGEN* вызывает *CoCreateGuid* и преобразует полученный GUID в один из четырех форматов, удобных для включения в исходный код на C++ или IDL. При работе в IDL используется четвертый формат (каноническая текстовая форма).

Чтобы связать физическое имя интерфейса с его определением на IDL, используется второй обязательный атрибут интерфейса – *[uuid]* . Атрибут *[uuid]* содержит один параметр – каноническую текстовую форму

```
GUID: [object, uuid(BDA4A270-A1BA-11dO-8C2C-0080C73925BA)]
interface ICalculator : IBaseInterface

HRESULT Clear(void);
HRESULT Add([in] long n);
HRESULT Sum([out, retval] long *pn);
```

При использовании при программировании на C или C++ физического имени интерфейса IID данного интерфейса представляет собой просто логическое имя интерфейса, предшествуемое префиксом IID_. Например, интерфейс ICalculator будет иметь IID, которым можно программно манипулировать, используя сгенерированную IDL константу IID_ICalculator. Для предотвращения коллизий между символическими именами интерфейсов можно использовать пространство имен C++.

Поскольку лишь немногие из компиляторов C++ могут поддерживать 128-битные числа, COM определяет C-структуру для представления 128-битовой величины GUID и предлагает псевдонимы для типов IID и CLSID с использованием следующего определения типов:

typedef struct *GUID*

DWORD Data1;
WORD Data2;
WORD Data3;
BYTE Data4[8];
GUID;
typedef GUID IID;
typedef GUID CLSID;

Внутренняя структура GUID для большинства программистов несущественна, так как единственная значимая операция, которую можно выполнить с GUID, – это проверка их эквивалентности. Для обеспечения эффективной передачи величин GUID как аргументов функций COM предусматривает также постоянные псевдонимы для ссылок (constant reference aliases) для каждого типа GUID:

#define REFGUID const GUID&
#define REFIID const IID&
#define REFCLSID const CLSID&

Чтобы иметь возможность сравнивать величины GUID, COM обеспечивает функции эквивалентности и перегружает операторы == и *!=* для постоянных ссылок GUID:

inline BOOL IsEqualGUID(REFGUID r1, REFGUID r2)

return !memcmp(&r1, &r2, sizeof(GUID));

#def1ne IsEqualIID(r1, r2) IsEqualGUID((r1) , (r2))
#define IsEqualCLSID(r1, r2) IsEqualGUID((r1), (r2))
inline BOOL operator == (REFGUID r1, REFGUID r2)

return !memcmp(&r1, &r2, sizeof(GUID));

inline BOOL operator != (REFGUID r1, REFGUID r2)

return !(r1 == r2);

Фактические заголовки SDK содержат условно компилируемые совместимые с C версии определений типа, макросов и встраиваемых функций, как показано выше.

Поскольку показано, что представления имен интерфейсов на этапе выполнения являются GUID, а не строками; это означает, что метод Dynamic_Cast, описанный в предыдущей главе, следует пересмотреть. Действительно, весь интерфейс IExtensibleObject должен быть изменен и преобразован в свой аналог IUnknown, совместимый с COM.

Интерфейс IUnknown

COM-интерфейс IUnknown имеет то же назначение, что и интерфейс IExtensibleObject, определенный в предыдущей главе. Последняя версия IExtensibleObject, появившаяся в конце предыдущей главы, имеет вид:

```
class IExtensibleObject

public:
virtual void *Dynamic_Cast(const char* pszType) = 0;
virtual void DuplicatePointer(void) = 0;
virtual void DestroyPointer(void) = 0;
```

Для определения типа на этапе выполнения был применен метод Dynamic_Cast, аналогичный оператору C++ dynamic_cast. Для извещения объекта о том, что указатель интерфейса дублировался, использовался метод DuplicatePointer. Для сообщения объекту, что указатель интерфейса уничтожен и все используемые им ресурсы могут быть освобождены, был применен метод DestroyPointer. Вот как выглядит определение IUnknown на C++:

```
extern "C" const IID IID_IUnknown: interface IUnknown

virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, void **ppv) = 0;
virtual ULONG STDMETHODCALLTYPE AddRef(void) = 0;
virtual ULONG STDMETHODCALLTYPE Release(void) = 0;
;
```

Заголовочные файлы SDK дают псевдоним interface ключевому слову C++ struct, используя препроцессор C. Поскольку интерфейсы в COM определены не как классы, а как структуры, то для того, чтобы сделать методы интерфейса общедоступными, ключевое слово public не требуется. Чтобы создать для целевой платформы COM-совместимые стековые фреймы, необходим макрос STDMETHODCALLTYPE. Если целевыми являются платформы Win32, то при использовании компилятора Microsoft C++ этот макрос раскрывается в _stdcall.

IUnknown функционально эквивалентен IExtensibleObject. Метод QueryInterface используется для динамического определения типа и аналогичен C++-оператору dynamic_cast. Метод AddRef используется для сообщения объекту, что указатель интерфейса дублирован. Метод Release используется для сообщения объекту, что указатель интерфейса уничтожен и все ресурсы, которые объект поддерживал от имени клиента, могут быть отключены. Главное различие между IUnknown и интерфейсом, определенным в предыдущей главе, заключается в том, что IUnknown использует идентификаторы GUID, а не строки для идентификации типов интерфейса на этапе выполнения.

IDL-определение IUnknown можно найти в файле unknwn.idl из директории SDK, содержащей заголовочные файлы:

```
// unknwn.idl – system IDL file
// unknwn.idl – системный файл IDL
[ local, object, uuid (00000000-0000-0000-C000-000000000046) ] interface IUnknown

HRESULT QueryInterface([in] REFIID riid, [out] void **ppv);
ULONG AddRef(void); ULONG Release(void);
```

Атрибут local подавляет генерирование сетевого кода для этого интерфейса. Этот атрибут необходим для того, чтобы смягчить требования COM о том, что все методы при вызове с удаленных машин должны возвращать HRESULT. Как будет показано в следующих главах, интерфейс IUnknown трактуется особым образом при работе с удаленными

объектами. Заметим, что фактические, то есть использующиеся на практике IDL-описания интерфейсов, которые содержатся в заголовках SDK, немного отличаются от определений, данных в этой книге. Фактические определения часто содержат дополнительные атрибуты для оптимизации генерируемого сетевого кода, которые не имеют отношения к нашему обсуждению. В случае сомнений обратитесь за полными определениями к последней версии заголовочных файлов SDK.

Интерфейс IUnknown является родительским для всех COM-интерфейсов. IUnknown – единственный интерфейс COM, который не наследует от другого интерфейса. Любой другой допустимый интерфейс COM должен быть прямым потомком IUnknown или какого-нибудь другого допустимого интерфейса COM, который, в свою очередь, должен сам наследовать или прямо от IUnknown, или от какого-нибудь другого допустимого интерфейса COM. Это означает, что на двоичном уровне все интерфейсы COM являются указателями на таблицы vtbl, которые начинаются с трех точек входа: QueryInterface, AddRef и Release. Все специфические для интерфейсов методы будут иметь точки входа в vtbl, которые появляются после этих трех общих точек входа.

Чтобы наследовать от интерфейса IDL, нужно или определить базовый интерфейс в том же IDL-файле, или использовать директиву import, чтобы сделать внешнее IDL-определение базового интерфейса явным в данной области действия:

```
// calculator.idl
[object, uuid(BDA4A270-A1BA-11dO-8C2C-0080C73925BA)]
interface ICalculator : IUnknown

import «unknwn.idl»;
// bring in def. of IUnknown
// импортируем определение IUnknown
HRESULT Clear(void);
HRESULT Add([in] long n);
HRESULT Sum([out, retval] long *pn);
```

Оператор import может появляться или внутри определения интерфейса, как показано здесь, или предшествовать описанию интерфейса в глобальной области действия. В любом из этих случаев действия оператора import одинаковы, он может многократно импортировать один IDL-файл без всякого ущерба. Поскольку сгенерированный C/C++ заголовочный файл будет требовать C/C++-версии импортируемого IDL-файла, чтобы обеспечить наследование, оператор import из IDL-файла будет транслирован в команду #include в генерируемом заголовочном C/C++-файле:

```
// calculator.h – generated by MIDL
// calculator.h – генерированный MIDL
// bring in def. of IUnknown
// вводим определения IUnknown
#include «unknwn.h»
extern "C" const IID IID_ICalculator;
interface ICalculator : public IUnknown

virtual HRESULT STDMETHODCALLTYPE Clear(void) = 0;
virtual HRESULT STDMETHODCALLTYPE Add(long n) = 0;
virtual HRESULT STDMETHODCALLTYPE Sum(long *pn) = 0;
```

Компилятор MIDL также создаст C-файл, содержащий фактические определения всех

GUID, имеющихся в исходном IDL-файле:

```
// calculator_i.c – generated by MIDL
const IID IID_ICalculator =
 0xBDA4A270, 0xA1BA, 0x11d0,  0x8C, 0x2C,
0x00, 0x80, 0xC7, 0x39, 0x25, 0xBA  ;
```

Каждый проект, который будет использовать этот интерфейс, должен или добавить calculator_i.c к своему файлу сборки (makefile), или включить calculator_i.c в один из исходных файлов на C или C++ с использованием препроцессора C. Если это не сделано, то идентификатору IID_ICalculator не будет выделено памяти для его 128-битного значения и проект не будет скомпонован по причине неразрешенных внешних идентификаторов.

COM не накладывает никаких ограничений на глубину иерархии интерфейсов при условии, что конечным базовым интерфейсом является IUnknown. Нижеследующий IDL является вполне допустимым и корректным для COM:

```
import «unknwn.idl»;
[object, uuid(DF12E151-A29A-11d0-8C2D-0080C73925BA)]
interface IAnimal : IUnknown
HRESULT Eat(void);

[object, uuid(DF12E152-A29A-11d0-8C2D-0080C73925BA)]
interface ICat : IAnimal

HRESULT IgnoreMaster(void);

[object, uuid(DF12E153-A29A-11d0-8C2D-0080C73925BA)]
interface IDog : IAnimal

HRESULT Bark(void);

[object, uuid(DF12E154-A29A-11d0-8C2D-0080C73925BA)]
interface IPug : IDog

HRESULT Snore(void);

[object, uuid(DF12E155-A29A-11d0-8C2D-0080C73925BA)]
interface IOldPug : IPug

HRESULT SnoreLoudly(void);
```

COM накладывает одно ограничение на наследование интерфейсов: интерфейсы COM не могут быть прямыми потомками более чем одного интерфейса. Следующий фрагмент в COM недопустим:

```
[object, uuid(DF12E156-A29A-11d0-8C2D-0080C73925BA)]
interface ICatDog : ICat, IDog

// illegal, multiple bases
// неверно, несколько базовых интерфейсов
HRESULT Meowbark(void);
```

COM запрещает наследование от нескольких интерфейсов по целому ряду причин. Одна из них состоит в том, что двоичное представление результирующего абстрактного базового класса C++ не будет независимым от компилятора. В этом случае COM уже не будет являться двоичным стандартом, независимым от разработчика. Другая причина кроется в тесной связи между COM и DCE RPC. При ограничении наследования интерфейсов одним базовым интерфейсом преобразование между интерфейсами COM и интерфейсными векторами DCE RPC вполне однозначно. В конце концов, отсутствие поддержки нескольких базовых интерфейсов не является ограничением, так как каждая реализация может выбрать для открытия столько интерфейсов, сколько пожелает. Это означает, что основанный на COM Cat/Dog по-прежнему допустим на уровне *реализации* :

```
class CatDog : public ICat, public IDog

//
...
;
```

Клиент, желающий трактовать объект как Cat/Dog, просто использует QueryInterface для привязки к объекту обоих типов указателей. Если один из вызовов QueryInterface не достигает успеха, то данный объект не является Cat/Dog и клиент может справляться с этим, как сумеет. Поскольку реализации могут открывать несколько интерфейсов, то запрет для интерфейсов наследовать более чем от одного интерфейса является лишь небольшой потерей в смысле семантической информации или информации о типе.

COM поддерживает стандарт обозначений, который показывает, какие интерфейсы доступны из объекта. Этот способ придерживается философии COM относительно отделения интерфейса от реализации и не раскрывает никаких деталей реализации объекта иначе, чем через список выставляемых им интерфейсов.

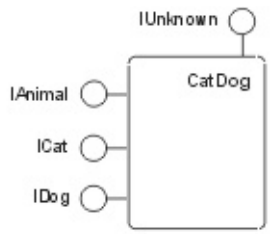

Рис. 2.4. Стандартные обозначения COM

Рисунок 2.4 показывает стандартное обозначение класса CatDog. Заметим, что из этой схемы можно сделать единственный вывод: если не произойдет катастрофических сбоев, объекты CatDog выставят четыре интерфейса: ICat, IDog, IAnimal и IUnknown.

Управление ресурсами и IUnknown

Как было в случае с *DuplicatePointer* и *DestroyPointer* из предыдущей главы, методы *AddRef* и *Release* из *IUnknown* имеют очень простой протокол, которого должны придерживаться все, кто пользуется указателями этих интерфейсов. Эти правила освобождают клиента от необходимости управлять временем жизни объекта, когда несколько интерфейсных указателей могут указывать или не указывать на один и тот же объект. Клиентам необходимо только следовать простым правилам *AddRef/Release* единообразно для всех интерфейсных указателей, с которыми им приходится сталкиваться, а объект будет сам управлять своим временем жизни.

Спецификация модели компонентных объектов (Component Object Model Specification)

содержит четкие определения правил подсчета ссылок COM. Понимание мотивировки этих определений имеет решающее значение при COM-программировании на C++. Эти правила COM о подсчете ссылок могут быть сведены к трем простым аксиомам:

Когда ненулевой указатель интерфейса копируется из одной ячейки памяти в другую, должен вызываться AddRef для извещения объекта о дополнительной ссылке.

Перед тем как произойдет перезапись той ячейки памяти, где содержится ненулевой указатель интерфейса, необходимо вызвать ***Release*** , чтобы известить объект, что ссылка уничтожается.

Избыточное количество вызовов ***AddRef*** и ***Release*** можно сократить, если иметь дополнительную информацию о связях между двумя и более ячейками памяти.

Аксиома о дополнительной информации введена главным образом для того, чтобы ввести возможность преобразования запутанных ситуаций в разумные и осмысленные идиомы программирования (например, стеки временных вызовов и сгенерированное компилятором занесение переменной в регистр не нуждаются в подсчете ссылок). Можно провести месяцы в поиске особых связей между переменными, содержащими явные указатели на интерфейс в программе и оптимизировать избыточные вызовы ***AddRef*** и ***Release*** , но поступать так было бы неосмотрительно. Выгода от удаления этих избыточных вызовов явно незначительна, так как даже в худшем случае, когда объект вызывается с расстояния более 8500 миль со средней скоростью передачи 14.4 кбит/сек, эти избыточные вызовы никогда не уйдут из вызывающего потока и нечасто требуют множество инструкций для выполнения.

Если руководствоваться приведенными выше тремя простыми аксиомами о подсчете ссылок в интерфейсных указателях, то можно записать это в виде руководящих принципов программирования, чтобы установить, когда вызывать и когда не вызывать ***AddRef*** и ***Release*** . Вот несколько типичных ситуаций, требующих вызова метода ***AddRef*** :

A1. Когда ненулевой интерфейсный указатель записывается в локальную переменную.

A2. Когда вызываемый объект пишет ненулевой интерфейсный указатель в параметр ***[out]*** или ***[in, out]*** метода или функции.

A3. Когда вызываемый объект возвращает ненулевой интерфейсный указатель как физический результат (physical result) функции.

A4. Когда ненулевой интерфейсный указатель пишется в элемент данных объекта.

Некоторые типичные ситуации, требующие вызова метода ***Release*** :

R1. Перед перезаписью ненулевой локальной переменной или элемента данных.

R2. Перед тем как покинуть область действия ненулевой локальной переменной.

R3. Когда вызываемый объект перезаписывает параметр ***[in,out]*** метода или функции, начальное значение которых отлично от нуля. Заметим, что параметры ***[out]*** предполагаются нулевыми при вводе и никогда не могут освобождаться вызываемым объектом.

R4. Перед перезаписью ненулевого элемента данных объекта.

R5. Перед завершением работы деструктора объекта, имеющего в качестве элемента данных ненулевой интерфейсный указатель.

Типичная ситуация, к которой применимо правило о дополнительной информации, возникает при передаче указателей интерфейсов функциям как параметрам ***[in]*** :

S1. Когда при вызове функции или метода ненулевой интерфейсный указатель передается через ***[in]*** -параметр, вызов ***AddRef*** и ***Release*** , не требуются, так как время жизни временной переменной в стеке является строгим подмножеством времени жизни выражения, использованного для инициализации формального аргумента.

Эти десять руководящих принципов охватывают ситуации, снова и снова возникающие при программировании в COM, и было бы неплохо их запомнить.

Чтобы конкретизировать правила подсчета ссылок в COM, предположим, что имеется глобальная функция, которая возвращает объекту интерфейсный указатель:

```
void GetObject([out] IUnknown **ppUnk);
```

и что имеется другая глобальная функция, которая выполняет некую полезную работу над объектом:

```
void UseObject([in] IUnknown *pUnk);
```

Написанный ниже код использует эти процедуры, чтобы управлять некоторыми объектами и возвращать интерфейсный указатель вызывающему объекту. Руководящие принципы, применимые к каждому оператору, указаны в комментариях к нему:

```
void GetAndUse(/* [out] */ IUnknown ** ppUnkOut)
 IUnknown *pUnk1 = 0, *pUnk2 = 0; *ppUnkOut =0;
// R3

// get pointers to one (or two) objects
// получаем указатели на один (или два) объекта
GetObject(&pUnk1);
//A2
GetObject(&pUnk2);
//A1
// set pUnk2 to point to first object
// устанавливаем pUnk2, чтобы указать на первый объект
if (pUnk2) pUnk2->Release():
//R1
if (pUnk2 = pUnk1) pUnk2->AddRef():
//A1
// pass pUnk2 to some other function
// передаем pUnk2 какой-нибудь другой функции
UseObject(pUnk2);
//S1
// return pUnk2 to caller using ppUnkOut parameter
// возвращаем pUnk2 вызывающему объекту, используя
// параметр ppUnkOut
if (*ppUnkOut = pUnk2) (*ppUnkOut)->AddRef();
// A2
// falling out of scope so clean up
// выходит за область действия и поэтому освобождаем
if (pUnk1) pUnkl->Release();
//R2
if (pUnk2) pUnk2->Release();
//R2
```

Важно отметить, что в вышеприведенном коде правило A2 применяется дважды, но по двум разным причинам. При вызове *GetObject* код выступает как вызывающий объект, а реализация *GetObject* является вызываемым объектом. Это означает, что реализация *GetObject* является ответственной за вызов *AddRef* через параметр *[out]* . При перезаписи памяти, на которую ссылается *ppUnkOut* , код выступает как вызываемый объект и корректно вызывает *AddRef* через интерфейсный указатель перед возвратом управления вызывающему объекту.

Существуют некоторые тонкости относительно *AddRef* и *Release* , подлежащие обсуждению. Как *AddRef* , так и *Release* предназначались для возврата 32-битного целого

числа без знака. Это целое число отражает общее количество оставшихся ссылок *после* применения операций ***AddRef*** или ***Release*** . Однако по целому ряду причин, связанных с многопоточным режимом, удаленным доступом и мультипроцессорной архитектурой, нельзя быть уверенным в том, что эта величина будет точно отражать общее число неосвобожденных интерфейсных указателей, и клиенту следует игнорировать ее, если только она не используется в целях диагностики при отладке.

Единственный случай, заслуживающий внимания, это когда ***Release*** возвращает нуль. Нулевой результат от ***Release*** надежно свидетельствует о том, что данный объект более не действителен ни в каком смысле. Однако обратное неверно. Это значит, что когда ***Release*** возвращает не нуль, нельзя утверждать, что объект еще работоспособен. Фактически, если ***Release*** был вызван указателем интерфейса столько же раз, сколько этим же указателем интерфейса был вызван ***AddRef*** , то данный указатель интерфейса недействителен и более не обеспечивает указание на действующий объект. В то же время возможно, что это – случайность, а объект все еще работоспособен благодаря другим, еще не освобожденным, указателям, и все может измениться в самый неподходящий момент. Чтобы однажды освобожденные (released) интерфейсные указатели более не использовались, можно, например, обнулять их сразу же после вызова метода ***Release*** :

```
inline void SafeRelease(IUnknown * &rpUnk)

if (rpUnk)

rpUnk->Release();
rpUnk = 0;
// rpUnk passed by reference
// rpUnk, переданный ссылкой
```

Когда этот способ применен, любое использование указателя интерфейса после его высвобождения немедленно вызовет ошибку доступа. Эта ошибка затем может быть достоверно воспроизведена и, можно надеяться, отловлена еще на этапе разработки.

Еще одна тонкость, относящаяся к ***AddRef*** и ***Release*** , состоит в выходе из блока. Функция ***GetAndUse*** , приведенная ранее, имеет только одну точку выхода. Это означает, что операторы, высвобождающие указатели интерфейса в конце функции, будут всегда выполняться ранее завершения работы функции. Если же функция завершит работу, не доходя до этих операторов – либо благодаря явному оператору ***return*** или же, что хуже, необработанному (unhandled) исключению C++, – то эти завершающие операторы будут пропущены и все ресурсы, удерживаемые неосвобожденными интерфейсными указателями, будут утеряны до окончания клиентской программы. Это означает, что к указателям интерфейса COM следует относиться с осторожностью, особенно при использовании их в средах, использующих исключения C++. Впрочем, это касается и других системных ресурсов, с которыми приходится работать, будь то семафоры или динамически распределяемая память. Далее в этой главе обсуждаются интеллектуальные COM-указатели, которые обеспечивают вызов ***Release*** во всех ситуациях.

Приведение типов и IUnknown

В предыдущей главе обсуждалось, почему необходимо определять тип на этапе выполнения в динамически собранной системе. Язык C++ предусматривает разумный механизм для динамического определения типа с помощью оператора dynamic_cast. Хотя эта

языковая возможность имеет собственную реализацию для каждого компилятора, в предыдущей главе было предложено средство урегулирования этого – добавление к каждому интерфейсу явного метода, являющегося семантическим эквивалентом dynamic_cast. Ниже приводится IDL-описание QueryInterface:

```
HRESULT QueryInterface([in] REFIID riid, [out] void **ppv);
```

Первый параметр (riid) является физическим именем запрошенного интерфейса. Второй параметр (ppv) указывает на переменную интерфейсного указателя, которая в случае успешного завершения будет содержать запрошенный указатель на интерфейс.

В ответ на запрос QueryInterface, если объект не поддерживает запрошенный тип интерфейса, он должен возвратить E_NOINTERFACE после установки *ppv в нулевое значение. Если же объект поддерживает запрошенный интерфейс, он должен перезаписать *ppv указателем запрошенного типа и возвратить HRESULT S_OK. Поскольку ppv является [out]-параметром, реализация QueryInterface должна выполнить AddRef для возвращаемого указателя перед тем, как вернуть управление вызывающему объекту (см. в этой главе выше руководящий принцип A2). Этот вызов AddRef должен быть согласован с вызовом Release со стороны клиента. Следующий код показывает динамическое определение типа с использованием оператора C++ dynamic_cast на примере иерархии типов Dog/Cat, описанного ранее в данной главе:

```
void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk)

IPug *pPug = 0;
pPug = dynamic_cast<IPug*> (pUnk);
if (pPug)
// the object is Pug-compatible
// объект совместим с Pug
pPug->Snore();
ICat *pCat = 0;
pCat = dynamic_cast<ICat*>(pUnk);
if (pCat)
// the object is Cat-compatible
// объект совместим с Cat
pCat->IgnoreMaster();
```

Если объект, переданный этой функции, совместим одновременно с ICat и с IDog, то задействованы обе функциональные возможности. Если же объект в действительности не совместим с ICat или с IDog, то данная функция просто проигнорирует пропущенный аспект объекта (или оба аспекта сразу). Ниже показан семантически эквивалентный вариант с использованием QueryInterface:

```
void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk)

HRESULT hr;
IPug *pPug = 0;
hr = pUnk->QueryInterface(IID_IPug, (void**)&pPug);
if (SUCCEEDED(hr))

// the object is Pug-compatible
// объект совместим с Pug
```

```
pPug->Snore();
pPug->Release();
// R2

ICat *pCat = 0;
hr = pUnk->QueryInterface(IID_ICat, (void**)&pCat);
if (SUCCEEDED(hr))

// the object is Cat-compatible
// объект совместим с Cat
pCat->IgnoreMaster();
pCat->Release(); // R2
```

Хотя имеются очевидные различия в синтаксисе, единственная существенная разница между двумя приведенными фрагментами кода состоит в том, что вариант, основанный на QueryInterface, подчиняется правилам подсчета ссылок COM.

Есть несколько тонкостей, связанных с QueryInterface и его употреблением. Метод QueryInterface может возвращать указатели только на тот же самый COM-объект, для которого он вызван. Глава 4 посвящена объяснению каждого нюанса этого оператора. Полезно, однако, отметить уже сейчас, что клиент не должен трактовать AddRef и Release как операции с *объектом*. Вместо этого следует рассматривать их как операции с *указателем интерфейса*. Это означает, что нижеследующий код ошибочен:

```
void BadCOMCode(/*[in]*/ IUnknown *pUnk)

ICat *pCat = 0;
IPug *pPug = 0;
HRESULT hr;
hr = pUnk->QueryInterface(IID_ICat, (void**)&pCat);
if (FAILED(hr)) goto cleanup;
hr = pUnk->QueryInterface(IID_IPug, (void**)&pPug);
if (FAILED(hr)) goto cleanup;
pPug->Bark();
pCat->IgnoreMaster();
cleanup:
if (pCat) pUnk->Release();
// pCat got AddRefed in QI
// pCat получил AddRef в QI
if (pPug) pUnk->Release();
// pDog got AddRefed in QI
// pDog получил AddRef в QI
```

Несмотря на то что все три указателя: pCat, pPug и pUnk – указывают на тот же самый объект, клиент не имеет права компенсировать AddRef, который происходит для pCat и pPug при вызове QueryInterface, вызовами Release для pUnk. Правильный вариант этого кода такой:

```
cleanup:
if (pCat) pCat->Release();
```

```
// use AddRefed ptr
// используем указатель AddRef
if (pPug) pPug->Release();
// use AddRefed ptr
// используем указатель AddRef
```

Здесь Release вызывается для того же *интерфейсного указателя*, для которого и AddRef (что произошло неявно, когда указатель был возвращен из QueryInterface). Это требование предоставляет разработчику значительную гибкость при реализации объекта. Например, объект может решить подсчитывать ссылки на каждый интерфейс, чтобы активным образом использовать ресурсы, которые обычно используются одним определенным интерфейсом на объект.

Еще одна тонкость относится ко второму параметру QueryInterface, имеющему тип void**. Весьма забавно то, что QueryInterface, являющийся основой системы типов COM, имеет довольно сомнительный в смысле типа аналог в C++:

```
HRESULT _stdcall QueryInterface(REFIID riid, void** ppv);
```

Как было отмечено ранее, клиенты вызывают QueryInterface, передавая объекту указатель на интерфейсный указатель в качестве второго параметра вместе с IID, который определяет тип ожидаемого интерфейсного указателя:

```
IPug *pPug = 0; hr = punk->QueryInterface(IID_IPug, (void**)&pPug);
```

К сожалению, для компилятора C++ таким же правильным выглядит и следующее:

```
IPug *pPug = 0; hr = punk->QueryInterface(IID_ICat, (void**)&pPug);
```

Даже еще более хитроумный вариант компилируется без ошибок:

```
IPug *pPug = 0; hr = punk->QueryInterface(IID_IPug, (void**)pPug);
```

Исходя из того, что правила наследования неприменимы к указателям, такое альтернативное определение QueryInterface не облегчает проблему:

```
HRESULT QueryInterface(REFIID riid, IUnknown** ppv);
```

так как неявное приведение типа к родительскому типу (upcasting) применимо только к объектам и указателям на объекты, а не к указателям на указатели на объекты:

```
IDerived **ppd; IBase **ppb = ppd;
// illegal
// неверно
```

То же ограничение применимо в равной мере и к ссылкам на указатели. Следующее альтернативное определение вряд ли более удобно для использования клиентами:

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

так как позволяет клиентам отказаться от приведения типа (cast). К сожалению, это решение не уменьшает количества ошибок (обе из предшествующих ошибок все еще возможны), а устраняя необходимость приведения, уничтожает и видимый индикатор того,

что устойчивость типов C++ может оказаться в опасности. Если желательна семантика QueryInterface, то выбор типов аргументов, сделанный корпорацией Microsoft, по крайней мере, разумен, если не надежен или изящен. Простейший путь избежать ошибок, связанных с QueryInterface,– это всегда быть уверенным в том, что IID соответствует типу указателя интерфейса, который проходит как второй параметр QueryInterface. На самом деле первый параметр QueryInterface описывает «форму» типа указателя второго параметра. Их связь может быть усилена на этапе компиляции с помощью такого макроса предпроцессора C:

```
#define IID_PPV_ARG(Type, Expr) IID_##type,
reinterpret_cast<void**>(static_cast<Type **>(Expr))
```

С помощью этого макроса[8] компилятор будет уверен в том, что выражение, использованное в приведенном ниже вызове QueryInterface, имеет правильный тип и что используется соответствующий уровень изоляции (indirecton):

```
IPug *pPug = 0; hr = punk->QueryInterface(IID_PPV_ARG(IPug, &pPug));
```

Этот макрос закрывает брешь, вызванную параметром void**, без каких-либо затрат на этапе выполнения.

Реализация IUnknown

Имея описанные выше образцы клиентского использования, легко видеть, как реализовать методы *IUnknown* . Примем предложенную выше иерархию типов *Dog/Cat* . Чтобы определить C++-класс, который реализует интерфейсы *IPug* и *ICat* , нужно просто добавить к списку базовых классов самые последние в иерархии наследования версии интерфейсов:

```
class PugCat : public IPug, public ICat
```

При использовании наследования компилятор C++ обеспечивает совместимость двоичного представления производного класса с каждым базовым классом. Для класса *PugCat* это означает, что все объекты *PugCat* будут содержать указатель *vptr* , указывающий на таблицу *vtbl* , совместимую с *IPug* . Объекты *PugCat* также будут содержать указатель *vptr* , указывающий на вторую таблицу *vtbl* , совместимую с *ICat* . Рисунок 2.5 показывает, как интерфейсы в качестве базовых классов соотносятся с представлением объектов.

Поскольку все функции-члены в COM-определениях интерфейса являются чисто виртуальными, производный класс должен обеспечивать реализацию каждого метода, имеющегося в любом из его интерфейсов. Методы, общие для двух или более интерфейсов (например, *QueryInterface* , *AddRef* и т. д.) нужно реализовывать только один раз, так как компилятор и компоновщик инициализируют все таблицы *vtbl* так, чтобы они указывали на одну реализацию метода. Таков естественный побочный эффект от использования множественного наследования в языке C++.

---

[8] 1 Который в значительной мере инспирирован дискуссией между автором и Tye McQueen во время семинара по COM.

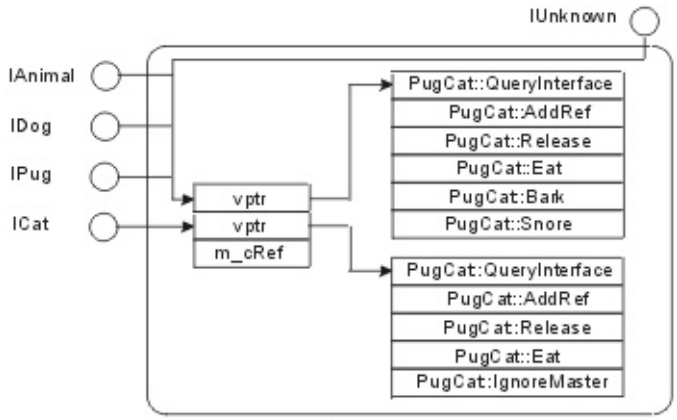

Рис. 2.5. PugCat

Следующий код является определением класса, которое создает объекты, поддерживающие интерфейсы *IPug* и *ICat* :

```
class PugCat : public IPug, public ICat

LONG mcRef;
protected:
virtual ~PugCat(void);
public: PugCat(void);
// IUnknown methods
// методы IUnknown
STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
STDMETHODIMP(ULONG) AddRef(void);
STDMETHODIMP(ULONG) Release(void);
// IAnimal methods
// методы IAnimal
STDMETHODIMP Eat(void);
// IDog methods
// методы IDog
STDMETHODIMP Bark(void);
// IPug methods
// методы IPug
STDMETHODIMP Snore(void);
// ICat methods
// методы ICat
STDMETHODIMP IgnoreMaster(void);
;
```

Отметим, что в классе должен быть реализован каждый метод, определенный в любом интерфейсе, от которого он наследует, так же, как и каждый метод, определенный в любых производных (implied) базовых интерфейсах (например, IDog, IAnimal). Для создания стековых фреймов, совместимых с COM, необходимо использовать макросы STDMETHODIMP и STDMETHODIMP. При ориентации на платформы Win32, использующие компилятор Microsoft C++, заголовки SDK определяют эти два макроса следующим образом:

```
#define STDMETHODIMP HRESULT stdcall
#define STDMETHODIMP(type) type stdcall
```

Заголовочные файлы SDK также определяют макросы *STDMETHOD* и STDMETHO*D* ,

которые можно использовать при определении интерфейсов без IDL-компилятора. В серийно выпускаемом программировании на COM эти два макроса не нужны.

Реализация AddRef и Release чрезвычайно прозрачна. Элемент данных mcRef отслеживает, сколько неосвобожденных интерфейсных указателей удерживают объект. Конструктор класса приводит счетчик ссылок в нулевое состояние:

```
PugCat::PugCat(void) : mcRef(0)
// initialize reference count to zero
// устанавливаем счетчик ссылок в нуль
```

Реализация *AddRef* в классе фиксирует путем увеличения счетчика ссылок, что вызывающий объект продублировал указатель интерфейса. Измененное значение счетчика ссылок возвращается для целей диагностики:

```
STDMETHODIMP(ULONG) AddRef(void)
 return ++mcRef;
```

Реализация *Release* фиксирует уничтожение указателя интерфейса простым уменьшением счетчика ссылок, а также производит соответствующее действие, когда счетчик ссылок достигает нуля. Для объектов, находящихся в динамически распределяемой области памяти, это означает вызов оператора *delete* для уничтожения объекта:

```
STDMETHODIMP(ULONG) Release(void)

LONG res = -mcRef;
if (res == 0) delete this;
return res;
```

Для кэширования обновленного счетчика ссылок необходимо использовать временную переменную, так как нельзя обращаться к элементам данных объекта после того, как объект уже уничтожен.

Заметим, что показанные реализации *Addref* и *Release* используют собственные операторы инкремента и декремента (увеличения и уменьшения на единицу). Для простой реализации это весьма разумно, так как COM не допускает более одного потока для обращения к объекту до тех пор, пока конструктор не обеспечит явный многопоточный доступ (почему и как конструктор сделает это, подробно описано в главе 5). В случае объектов, доступных в многопоточной среде, для автоматического подсчета ссылок следует использовать подпрограммы Win32 *InterlockedIncrement/InterlockedDecrement* :

```
STDMETHODIMP(ULONG) AddRef(void)

return InterlockedIncrement(&mcRef);

STDMETHODIMP(ULONG) Release(void)

LONG res = InterlockedDecrement(&mcRef);
if (res == 0) delete this; return res;
```

Этот код несколько менее эффективен, чем версии, использующие собственные

операторы C++. Но, вообще говоря, разумнее использовать менее эффективные варианты *InterlockedIncrement / InterlockedDecrement* , так как известно, что они надежны во всех ситуациях и освобождают разработчика от необходимости сохранять две версии практически одинакового кода.

Показанные выше реализации *AddRef* и *Release* предполагают, что объект может размещаться только в динамически распределяемой области памяти (в «куче») с использованием C++-оператора *new* . В определении класса деструктор сделан защищенной операцией для обеспечения того, чтобы ни один экземпляр класса не был определен никаким другим способом. Однако иногда желательно иметь объекты, не размещенные в «куче». Для этих объектов вызов *delete* в последнем вызове *Release* был бы гибельным. Так как единственной причиной для того, чтобы объект в первую очередь поддерживал счетчик ссылок, была необходимость вызова *delete this* , допустимо оптимизировать счетчик ссылок для объектов, не содержащихся в динамически распределяемой области памяти:

```
STDMETHODIMP(ULONG) GlobalVar::AddRef(void)

return 2;
// any non-zero value is legal
// допустима любая ненулевая величина

STDMETHODIMP(ULONG) GlobalVar::Release (void)

return 1;
// any non-zero value is legal
// допустима любая ненулевая величина
```

Эта реализация использует тот факт, что результаты *AddRef* и *Release* служат только для сведения и не обязаны быть точными.

При наличии реализации *AddRef* и *Release* единственным еще не реализованным методом из *IUnknown* остается *QueryInterface* . Его реализации должны отслеживать иерархию типов объекта и использовать статические приведения типов для возврата правильного типа указателя для всех поддерживаемых интерфейсов. Для определения класса *PugCat* , рассмотренного ранее, следующий код является корректной реализацией *QueryInterface* : STDMETHODIMP

```
PugCat::QueryInterface(REFIID riid, void **ppv)

assert(ppv != 0);
// or return EPOINTER in production
// или возвращаем EPOINTER в реальный продукт
if (riid == IIDIPug) *ppv = staticcast<IPug*>(this);
else if (riid == IIDIDog) *ppv = staticcast<IDog*>(this);
else if (riid == IIDIAnimal)
// cat or pug?
// кот или мопс?
*ppv == staticcast<IDog*>(this);
else if (riid == IIDIUnknown)
// cat or pug?
// кот или мопс?
*ppv = staticcast<IDog*>(this);
else if (riid == IIDICat) *ppv = staticcast<ICat*>(this);
```

```
else

// unsupported interface
// неподдерживаемый интерфейс
*ppv = 0;
return ENOINTERFACE;

// if we reach this point, *ppv is non-null
// and must be AddRef'ed (guideline A2)
// если мы дошли до этого места, то *ppv ненулевой
// и должен быть обработан AddRef'ом ( принцип A2)
reinterpretcast<IUnknown*>(*ppv)->AddRef();
return SOK;
```

Использование static*cast* более предпочтительно, чем традиционные приведения типа в стиле C:

```
*ppv = (IPug*)this;
```

так как вариант static*cast* вызовет ошибку этапа компиляции, если произведенное приведение типа не согласуется с существующим базовым классом.

Заметим, что в показанной здесь реализации *QueryInterface* при запросе на интерфейс, поддерживающийся более чем одним базовым интерфейсом (например, *IUnknown* , *IAnimal*) приведение типа должно явно выбрать более определенный базовый класс. Например, для класса *PugCat* такой вполне безобидно выглядящий код не откомпилируется:

```
if (riid == IIDIUnknown) *ppv = staticcast<IUnknown*>(this);
```

Этот код не пройдет компиляцию, поскольку такое приведение типа является неоднозначным и может соответствовать более чем одному базовому классу. Это было показано в случае *FastString* и *IExtensibleObject* из предыдущей главы. Вместо этого реализация должна более точно выбрать тип для приведения:

```
if (riid == IIDIUnknown) ppv = staticcast<IDog*>(this);
или if (riid == IIDIUnknown) ppv = staticcast<ICat*>(this);
```

Каждый из этих двух фрагментов кода допустим для реализации *PugCat* . Первый вариант предпочтительнее, так как многие компиляторы выдают несколько более эффективный код, когда использован крайний левый базовый класс[9].

Использование указателей интерфейса COM

Программисты C++ должны использовать методы *IUnknown* явно, потому что перевод модели COM на язык C++ не предусматривает использования среды поддержки выполнения (runtime layer) между кодом клиента и кодом объекта. Поэтому *IUnknown* можно рассматривать просто как набор обещаний, которые все программисты COM дают друг

9 1 Который в значительной мере инспирирован дискуссией между автором и Tye McQueen во время семинара по COM.

другу. Это дает преимущество программистам C++, так как C++ может создавать код, который потенциально более эффективен, чем языки, которые требуют такого динамического слоя при работе с COM.

При работе на Visual Basic и Java, в отличие от C++, программисты никогда не видят *QueryInterface* , *AddRef* или *Release* . Для этих двух языков детали *IUnknown* надежно скрыты за поддерживающей эти языки виртуальной машиной. На Java *QueryInterface* просто отображается в приведение типа:

```
public void TryToSnoreAndIgnore(Object obj)

IPug pug;
try

pug = (IPug)obj;
// VM calls QueryInterface
// VM вызывает QueryInterface
pug.Snore();

catch (Throwable ex)

// ignore method or QI failures
// игнорируем сбой метода или QI

ICat cat;
try

cat = (ICat)obj;
// VM calls QueryInterface
// VM вызывает QueryInterface
cat.IgnoreMaster();

catch (Throwable ex)

// ignore method or QI failures
// игнорируется сбой метода или QI
```

Visual Basic не требует от клиентов приведения типов. Вместо этого, когда указатель интерфейса присваивается переменной неподходящего типа, виртуальная машина (VM) Visual Basic молча вызывает *QueryInterface* от имени клиента:

```
Sub TryToSnoreAndIgnore(obj as Object)
On Error Resume Next
' ignore errors
' игнорируем ошибки
Dim pug as IPug
Set pug = obj
' VM calls QueryInterface
' VM вызывает QueryInterface
If Not (pug is Nothing)
Then pug.Snore
```

```
End
if Dim cat as ICat
Set cat = obj
' VM calls QueryInterface
' VM вызывает QueryInterface
If Not (cat is Nothing)
Then cat.IgnoreMaster
End if End Sub
```

Обе виртуальные машины, как Java, так и Visual Basic, выбросят при сбое *QueryInterface* исключения. В обеих средах виртуальная машина автоматически преобразует языковую концепцию живучести переменной в явные вызовы *AddRef* и *Release* , избавляя клиента и от этой подробности.

Одна методика, потенциально способная упростить использование в COM интерфейсных указателей из C++, состоит в том, чтобы скрыть их в классе интеллектуальных указателей. Это устраняет необходимость необработанных (*raw*) вызовов методов *IUnknown* . В идеале интеллектуальный указатель COM будет:

Корректно обрабатывать каждый вызов *Add/Release* во время присваивания.

Автоматически уничтожать интерфейс в деструкторе, что снижает возможность утечки ресурса и повышает безопасность (надежность) исключений.

Использует систему типов C++ для упрощения вызовов *QueryInterface* .

Прозрачным образом (незаметно для пользователя или программы) замещает необработанные интерфейсные указатели в существующем коде *без компрометации правильности программы* .

Последний пункт представляет собой чрезвычайно серьезную проблему. Интернет забит интеллектуальными COM-указателями, которые проделывают прозрачную замену обычных указателей, но при этом вводят столько же скрытых ошибок, сколько претендуют устранить. Visual C++ 5.0, например, фактически действует с тремя такими указателями (один на MSC, другой на ATL, а третий для поддержки Direct-to-COM), которые очень просто использовать как правильно, так и неправильно. В сентябрьском 1995 года и в февральском 1996 года выпусках "*C++ Report* " опубликованы две статьи, где на примерах показаны различные подводные камни при использовании интеллектуальных указателей[10]. Исходный код, который приводится в данной книге, содержит интеллектуальный COM-указатель, созданный в процессе написания этих двух статей. В нем делается попытка учесть общие ошибки, случающиеся как в простых, так и в интеллектуальных указателях COM. Класс интеллектуальных указателей, *SmartInterface* , имеет два шаблонных (template) параметра: тип интерфейса в C++ и указатель на соответствующий *IID* . Все обращения к методам *IUnknown* скрыты путем перегрузки операторов:

```
#include «smartif.h»
void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk)

// copy constructor calls QueryInterface
// конструктор копирования вызывает QueryInterface
SmartInterface<IPug, &IIDIPug> pPug = pUnk;
if (pPug)
```

10 *1* Эти статьи можно найти на сайтах http:/www.develop.com/dbox/cxx/InterfacePtr.htm и http://www.develop.com/dbox/cxx/SmartPtr.htm.

```
// typecast operator returns null-ness
// оператор приведения типа возвращает нуль pPug->Snore();
// operator-> returns safe raw ptr
// оператор -> возвращает прямой указатель
// copy constructor calls QueryInterface
// конструктор копирования вызывает QueryInterface
SmartInterface<ICat, &IIDICat> pCat = pUnk;
if (pCat)
// typecast operator returns null-ness
// оператор приведения типа возвращает нуль pCat->IgnoreMaster();
// operator-> returns safe raw ptr
// оператор -> возвращает прямой указатель
// destructors release held pointers on leaving scope
// деструкторы освобождают удерживаемые указатели при
// выходе из области действия
```

Интеллектуальные указатели выглядят очень привлекательными на первый взгляд, но могут оказаться очень опасными, так как погружают программиста в дремотное состояние; будто бы ничего страшного, относящегося к COM, произойти не может. Интеллектуальные указатели действительно решают реальные проблемы, особенно связанные с исключениями; однако при неосторожном употреблении они могут внести столько же дефектов, сколько они предотвращают. Например, многие интеллектуальные указатели позволяют вызывать любой метод интерфейса через оператор интеллектуального указателя –> . К сожалению, это позволяет клиенту вызывать *Release* с помощью этого оператора-стрелки без сообщения базовому интеллектуальному указателю о том, что его автоматический вызов *Release* в его деструкторе теперь является излишним и недопустимым.

Оптимизация QueryInterface

Фактически реализация *QueryInterface* , показанная ранее в этой главе, очень проста и легко может поддерживаться любым программистом, имеющим хоть некоторое представление о COM и C++. Тем не менее, многие среды и каркасы приложений поддерживают реализацию, управляемую данными. Это помогает достичь большей расширяемости и эффективности благодаря уменьшению размера кода. Такие реализации предполагают, что каждый совместимый с COM класс предусматривает таблицу, которая отображает каждый поддерживаемый IID на какой-нибудь аспект объекта, используя фиксированные смещения или какие-то другие способы. В сущности, реализация QueryInterface , приведенная ранее в этой главе, строит таблицу, основанную на скомпилированном машинном коде для каждого из последовательных операторов if, а фиксированные смещения вычисляются с использованием оператора staticcast (staticcast просто добавляет смещение базового класса, чтобы найти совместимый с типом указатель vptr).

Чтобы реализовать управляемый таблицей *QueryInterface* , необходимо сначала определить, что эта таблица будет содержать. Как минимум, каждый элемент таблицы должен содержать указатель на *IID* и некое дополнительное состояние, которое позволит реализации найти указатель *vptr* объекта для запрошенного интерфейса. Хранение указателя функции в каждом элементе таблицы придаст этому способу максимальную гибкость, так как это даст возможность добавлять новые методики поиска интерфейсов к обычному вычислению смещения, которое используется для приведения к базовому классу. Исходный код в приложении к данной книге содержит заголовочный файл *inttable.h* , который

определяет элементы таблицы интерфейсов следующим образом:

```
// inttable.h (book-specific header file)
// inttable.h (заголовочный файл, специфический для этой книги)
// typedef for extensibility function
// typedef для функции расширяемости
typedef HRESULT (*INTERFACEFINDER) (void *pThis, DWORD dwData, REFIID riid, void **ppv);
// pseudo-function to indicate entry is just offset
// псевдофункция для индикации того, что запись просто
// является смещением
#define ENTRYISOFFSET INTERFACEFINDER(-1)
// basic table layout // представление базовой таблицы
typedef struct INTERFACEENTRY

const IID * pIID;
// the IID to match
// соответствующий IID
INTERFACEFINDER pfnFinder;
// функция finder DWORD dwData;
// offset/aux data
// данные по offset/aux
 INTERFACEENTRY;
```

Заголовочный файл также содержит следующие макросы для создания интерфейсных таблиц внутри определения класса:

```
// Inttable.h (book-specific header file)
// Inttable.h (заголовочный файл, специфический для данной книги)
#define BASEOFFSET(ClassName, BaseName) (DWORD(staticcast<BaseName*>(reinterpretcast<ClassName*>(0x10000000))) – 0x10000000)
#define BEGININTERFACETABLE(ClassName) typedef ClassName ITCls;const INTERFACEENTRY *GetInterfaceTable(void) static const INTERFACEENTRY table [] = \i0
#define IMPLEMENTSINTERFACE(Itf) &IID##Itf,ENTRYISOFFSET,BASEOFFSET(ITCls,Itf),
#define IMPLEMENTSINTERFACEAS(req, Itf) &IID##req,ENTRYISOFFSET, BASEOFFSET(ITCls, Itf),
#define ENDINTERFACETABLE() 0, 0, 0 ; return table;
```

Все, что требуется, – это стандартная функция, которая может анализировать интерфейсную таблицу в ответ на запрос *QueryInterface* . Такая функция содержится в файле *Inttable.h* :

```
// inttable.cpp (book-specific source file)
// inttable.h (заголовочный файл, специфический для данной книги)
HRESULT InterfaceTableQueryInterface(void *pThis, const INTERFACEENTRY *pTable, REFIID riid, void **ppv)

if (InlineIsEqualGUID(riid, IIDIUnknown))

// first entry must be an offset
// первый элемент должен быть смещением
```

```
*ppv = (char*)pThis + pTable->dwData;
((Unknown*) (*ppv))->AddRef () ;
// A2
return SOK;
 else

HRESULT hr = ENOINTERFACE;
while (pTable->pfnFinder)

// null fn ptr == EOT
if (!pTable->pIID || InlineIsEqualGUID(riid,*pTable->pIID))

if (pTable->pfnFinder == ENTRYISOFFSET)

*ppv = (char*)pThis + pTable->dwData;
((IUnknown*)(*ppv))->AddRef();
// A2
hr = SOK;
break;
 else

hr = pTable->pfnFinder(pThis, pTable->dwData, riid, ppv);
if (hr == SOK) break;


pTable++;

if (hr!= SOK)
*ppv = 0;
return hr;
```

Получив указатель на запрошенный объект, *InterfaceTableQueryInterface* сканирует таблицу в поисках элемента, соответствующего запрошенному *IID* , и либо добавляет соответствующее смещение, либо вызывает соответствующую функцию. Приведенный выше код использует усовершенствованную версию *IsEqualGUID* , которая генерирует несколько больший код, но результаты по скорости примерно на 20-30 процентов превышают данные по существующей реализации, которая не управляется таблицей. Поскольку код для *InterfaceTableQueryInterface* появится в исполняемой программе только один раз, это весьма неплохой компромисс.

Очень легко автоматизировать поддержку COM для любого класса C++, основанную на таком табличном управлении, простым использованием C-препроцессора. Следующий фрагмент из заголовочного файла *impunk.h* определяет *QueryInterface* , *AddRef* и *Release* для объекта, использующего интерфейсные таблицы и расположенного в динамически распределяемой области памяти:

```
// impunk.h (book-specific header file)
// impunk.h (заголовочный файл, специфический для данной книги)
// AUTOLONG is just a long that constructs to zero
// AUTOLONG – это просто long, с конструктором,
// устанавливающим значение в О
```

```
struct AUTOLONG

LONG value;
AUTOLONG (void) : value (0)
;

#define IMPLEMENTUNKNOWN(ClassName)
AUTOLONG mcRef;
STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
return InterfaceTableQueryInterface(this,
GetInterfaceTable(), riid, ppv);

STDMETHODIMP(ULONG) AddRef(void)
return InterlockedIncrement(&mcRef.value);

STDMETHODIMP(ULONG) Release(void)
ULONG res = InterlockedDecrement(&mcRef.value) ;
if (res == 0)
delete this;
return res;
```

Настоящий заголовочный файл содержит дополнительные макросы для поддержки объектов, не находящихся в динамически распределяемой области памяти.

Для реализации примера ***PugCat*** , уже встречавшегося в этой главе, необходимо всего лишь удалить текущие реализации ***QueryInterface*** , ***AddRef*** и ***Release*** и добавить соответствующие макросы:

```
class PugCat : public IPug, public ICat

protected:
virtual ~PugCat(void);
public: PugCat(void);
// IUnknown methods
// методы IUnknown
IMPLEMENTUNKNOWN (PugCat)
BEGININTERFACETABLE(PugCat)
IMPLEMENTSINTERFACE(IPug)
IMPLEMENTSINTERFACE(IDog)
IMPLEMENTSINTERFACEAS(IAnimal,IDog)
IMPLEMENTSINTERFACE(ICat)
ENDINTERFACETABLE()
// IAnimal methods
// методы IAnimal
STDMETHODIMP Eat(void);
// IDog methods
// методы IDog
STDMETHODIMP Bark(void);
// IPug methods
// методы IPug
STDMETHODIMP Snore(void);
// ICat methods
```

```
// методы ICat
STDMETHODIMP IgnoreMaster(void);
;
```

Когда используются эти макросы препроцессора, для поддержки *IUnknown* не требуется никакого дополнительного кода. Все, что осталось сделать, это реализовать текущие методы интерфейса, которые делают этот класс уникальным.

Типы данных

Все интерфейсы COM должны быть определены в IDL. IDL позволяет описывать довольно сложные типы данных в стиле, не зависящем от языка и платформы. Рисунок 2.6 показывает базовые типы, которые поддерживаются IDL, и их отображения в языки C, Java и Visual Basic. Целые и вещественные типы не требуют объяснений. Первые «интересные» типы данных, встречающиеся при программировании в COM, – это символы и строки.

Язык	IDL	Microsoft C++	Visual Basic	Microsoft Java
Основные типы	boolean	unsigned char	unsupported	char
	byte	unsigned char	unsupported	char
	small	char	unsupported	char
	short	short	Integer	short
	long	long	Long	int
	hyper	_int64	unsupported	long
	float	float	Single	float
	double	double	Double	double
	char	unsigned char	unsupported	char
	wchar_t	wchar_t	Integer	short
	enum	enum	Enum	int
	Interface Pointer	Interface Pointer	Interface Ref.	Interface Ref.
Расширенные типы	VARIANT	VARIANT	Variant	ms.com.Variant
	BSTR	BSTR	String	java.lang.String
	VARIANT_BOOL	short [-1/0]	Boolean [True/False]	boolean [true/false]

Рис. 2.6. Базовые типы COM

Все символы в COM представлены с использованием типа данных OLECHAR. Для Windows NT, Windows 95, Win32s и Solaris OLECHAR – это просто typedef для типа данных C wchar_t. Специфика других платформ описана в соответствующих документациях. Платформы Win32 используют тип данных wchar_t для представления 16-битных символов Unicode[11]. Поскольку типы указателей в IDL созданы так, что указывают на одиночные переменные, а не на массивы, то IDL вводит атрибут [string], чтобы подчеркнуть, что

[11] 1 Тип OLECHAR был предпочтен типу данных TCHAR, используемому Wn32 API, чтобы избежать необходимости поддержки двух версий каждого интерфейса (CHAR и WCHAR). Поддерживая только один тип символов, разработчики объектов становятся независимыми от типов символов препроцессора UNICODE, который используется их клиентами.

указатель указывает на массив-строку с завершающим нулем:

```
HRESULT Method([in, string] const OLECHAR *pwsz);
```

Для определения строк и символов, совместимых с OLECHAR, в COM введен макрос OLESTR, который приписывает букву L перед строковой или символьной константой, информируя таким образом компилятор о том, что эта константа имеет тип wchar_t. Например, правильным будет такой способ инициализировать указатель OLECHAR с помощью строкового литерала:

```
const OLECHAR *pwsz = OLESTR(«Hello»);
```

Под Win32 или Solaris это эквивалентно

```
const wchar_t *pwsz = L"Hello";
```

Первый вариант предпочтительней, так как он будет надежно компилироваться на всех платформах.

Поскольку часто возникает необходимость копировать строки на основе типа wchar_t в обычные буфера на основе char, то динамическая библиотека C предлагает две процедуры для преобразования типов:

```
size_t mbstowcs(wchar_t *pwsz, const char *psz, int cch);
size_t wcstombs(char *psz, const wchar_t *pwsz, int cch);
```

Эти две процедуры работают аналогично динамической C-процедуре strncpy, за исключением того, что в эти процедуры как часть операции копирования включено расширение или сужение строки. Следующий код показывает, как параметр метода, размещенный в OLECHAR, можно скопировать в элемент данных, размещенный в char:

```
class BigDog : public ILabrador

char m_szName[1024] ;
public:
STDMETHODIMP SetName(/* [in,string]*/ const OLECHAR *pwsz)

HRESULT hr = S_OK;
size_t cb = wcstombs(m_szName, pwsz, 1024);
// check for buffer overflow or bad conversion
// проверяем переполнение буфера или неверное преобразование
if (cb == sizeof(m_szName) || cb == (size_t)-1)

m_szName[0] = 0; hr = E_INVALIDARG;

return hr;

;
```

Этот код является довольно простым, хотя программист должен осознавать, что используются два различных типа символов. Несколько более сложный (и чаще встречающийся) случай – преобразование между типами данных OLECHAR и TCHAR из Win32. Так как OLECHAR условно компилируется как char или wchar_t, то при реализации

метода необходимо должным образом рассмотреть оба сценария:

```
class BigDog : public ILabrador

TCHAR m_szName[1024];
// note TCHAR-based string
// отметим строку типа TCHAR
public:
STDMETHODIMP SetName( /*[in,string]*/ const OLECHAR *pwsz)

HRESULT hr = S_OK;
#ifdef UNICODE
// Unicode build (TCHAR == wchar_t)
// конструкция Unicode (TCHAR == wchar_t)
wcsncpy(m_szName, pwsz, 1024);
// check for buffer overflow
// проверка на переполнение буфера
if (m_szName[1023] != 0)

m_szName[0] = 0;
hr = E_INVALIDARG;

#else
// Non-Unicode build (TCHAR == char)
// не является конструкцией Unicode (TCHAR == char)
size_t cb = wcstombs(m_szName, pwsz, 1024);
// check for buffer overflow or bad conversion
// проверка переполнения буфера или ошибки преобразования
if (cb == sizeof(m_szName) || cb == (size_t)-1)

m_szName[0] =0;
hr = E_INVALIDARG;

#endif return hr;

;
```

Очевидно, операции с преобразованиями OLECHAR в TCHAR значительно сложнее. Но, к сожалению, это самый распространенный сценарий при программировании в COM на базе Win32.

Одним из подходов к упрощению преобразования текста является применение системы типов C++ и использование перегрузки функций для выбора нужной строковой процедуры, построенной на типах параметров. Заголовочный файл ustring.h из приложения к этой книге содержит семейство библиотечных строковых процедур, аналогичных стандартным библиотечным процедурам C, которые находятся в файле string.h. Например, функция strncpy имеет четыре соответствующих процедуры, зависящие от каждого из параметров, которые могут быть одного из двух символьных типов (wchar_t или char):

```
// from ustring.h (book-specific header)
// из ustring.h (заголовок, специфический для данной книги)
inline bool ustrncpy(char *p1, const wchar_t *p2, size_t c)
```

```
size_t cb = wcstombs(p1, p2, c);
return cb != c && cb != (size_t)-1;
;
inline bool ustrncpy(wchar_t *p1, const wchar_t *p2, size_t c)

wcsncpy(p1, p2, c);
return p1[c – 1] == 0;
;
inline bool ustrncpy(char *p1, const char *p2, size_t c)

strncpy(p1, p2, c);
return p1[c – 1] == 0;
;
inline bool ustrncpy(wchar_t *p1, const char *p2, size_t c)

size_t cch = mbstowcs(p1, p2, c);
return cch != c && cch != (size_t)-1;
```

Отметим, что для любого сочетания типов идентификаторов может быть найдена соответствующая перегруженная функция ustrncpy, причем результат показывает, была или нет вся строка целиком скопирована или преобразована. Поскольку эти процедуры определены как встраиваемые (inline) функции, их использование не внесет никаких затрат при выполнении. С этими процедурами предыдущий фрагмент кода станет значительно проще и не потребует условной компиляции:

```
class BigDog : public ILabrador

TCHAR m_szName[1024];
// note TCHAR-based string
// отметим строку типа TCHAR
public:
STDMETHODIMP SetName(/* [in,string] */ const OLECHAR *pwsz)

HRESULT hr = S_OK;
// use book-specific overloaded ustrncpy to copy or convert
// используем для копирования и преобразования
// перегруженную функцию ustrncpy, специфическую для данной книги
if (!ustrncpy(m_szName, pwsz, 1024))

m_szName[0] = 0;
hr = E_INVALIDARG;
 return hr;

;
```

Соответствующие перегруженные функции для процедур strlen, strcpy и strcat также включены в заголовочный файл ustring.h.

Использование перегрузки библиотечных функций для копирования строк из одного буфера в другой, как это показано выше, обеспечивает лучшее качество исполнения, уменьшает размер кода и непроизводительные издержки программиста. Однако часто возникает ситуация, когда одновременно используются COM и API-функции Win32, что не

дает возможности применить эту технику. Рассмотрим следующий фрагмент кода, читающий строку из элемента редактирования и преобразующий ее в IID:

```
HRESULT IIDFromHWND(HWND hwnd, IID& riid)

TCHAR szEditText[1024];
// call a TCHAR-based Win32 routine
// вызываем TCHAR-процедуру Win32
GetWindowText(hwnd, szEditText, 1024);
// call an OLECHAR-based COM routine
// вызываем OLECHAR-процедуру COM
return IIDFromString(szEditText, &riid);
```

Допуская, что этот код скомпилирован с указанным символом С-препроцессора UNICODE; он работает безупречно, так как TCHAR и OLECHAR являются просто псевдонимами wchar_t и никакого преобразования не требуется. Если же функция скомпилирована с версией Win32 API, не поддерживающей Unicode, то TCHAR является псевдонимом для char, и первый параметр для IIDFromString имеет неправильный тип. Чтобы решить эту проблему, нужно провести условную компиляцию:

```
HRESULT IIDFromHWND(HWND hwnd, IID& riid)

TCHAR szEditText[1024];
GetWindowText(hwnd, szEditText, 1024);
#ifdef UNICODE return IIDFromString(szEditText, &riid);
#else OLECHAR wszEditText[l024];
ustrncpy(wszEditText, szEditText, 1024);
return IIDFromString(wszEditText, &riid);
#endif
```

Хотя этот фрагмент и генерирует оптимальный код, очень утомительно применять эту технику всякий раз, когда символьный параметр имеет неверный тип. Можно справиться с этой проблемой, если использовать промежуточный (shim) класс с конструктором, принимающим в качестве параметра любой тип символьной строки. Этот промежуточный класс должен также содержать в себе операторы приведения типа, что позволит использовать его в обоих случаях: когда ожидается const char * или const wchar_t *. В этих операциях приведения промежуточный класс либо выделяет резервный буфер и производит необходимое преобразование, либо просто возвращает исходную строку, если преобразования не требовалось. Деструктор промежуточного класса может затем освободить все выделенные буферы. Заголовочный файл ustring.h содержит два таких промежуточных класса: _U и _UNCC. Первый предназначен для нормального использования; второй используется с функциями и методами, тип аргументов которых не включает спецификатора const[12] (таких как IIDFromString). При возможности применения двух промежуточных классов предыдущий фрагмент кода может быть значительно упрощен:

---

[12] 2 _UNCC является просто версией _U и имеет операторы приведения типа для wchart * и char *. Хотя расширенный вариант можно использовать где угодно, автор предпочитает использовать его только при согласовании с непостоянно корректными интерфейсами, чтобы подчеркнуть, что система типов в некоторой степени компрометируется. Увы, многие из COM API не являются постоянно корректными, так что промежуточный класс _UNCC применяется очень часто.

```
HRESULT IIDFromHWND(HWND hwnd, IID& riid)

TCHAR szEditText[1024];
GetWindowText(hwnd, szEditText, 1024);
// use _UNCC shim class to convert if necessary
// используем для преобразования промежуточный класс _UNCC,
// если необходимо
return IIDFromString(_UNCC(szEditText), &riid);
```

Заметим, что не требуется никакой условной компиляции. Если код скомпилирован с версией Win32 с поддержкой Unicode, то класс _UNCC просто пропустит исходный буфер через свой оператор приведения типа. Если же код компилируется с версией Win32, не поддерживающей Unicode, то класс _UNCC выделит буфер и преобразует строку в Unicode. Затем деструктор _UNCC освободит буфер, когда операция будет выполнена полностью[13].

Следует обсудить еще один дополнительный тип данных, связанный с текстом, – BSTR. Строковый тип BSTR нужно применять во всех интерфейсах, которые предполагается использовать из языков Visual Basic или Java. Строки BSTR являются OLECHAR-строками с префиксом длины (length-prefix) в начале строки и нулем в ее конце. Префикс длины показывает число *байт* , содержащихся в строке (исключая завершающий нуль) и записан в форме четырехбайтового целого числа, непосредственно предшествующего первому символу строки. Рисунок 2.7 демонстрирует BSTR на примере строки «Hi». Чтобы позволить методам свободно возвращать строки BSTR без заботы о выделении памяти, все BSTR размещены с помощью распределителя памяти, управляемого COM. В COM предусмотрено несколько API-функций для управления BSTR:

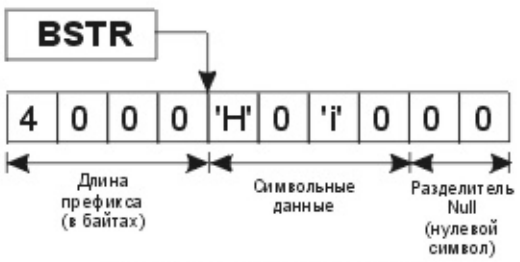

Рис. 2.7. Строка "Hi" как BSTR

```
// from oleauto.h
// allocate and initialize a BSTR
// выделяем память и инициализируем строку BSTR
BSTR SysAllocString(const OLECHAR *psz);
BSTR SysAllocStringLen(const OLECHAR *psz, UINT cch);
// reallocate and initialize a BSTR
// повторно выделяем память и инициализируем BSTR
INT SysReAllocString(BSTR *pbstr, const OLECHAR *psz);
INT SysReAllocStringLen(BSTR *pbstr, const OLECHAR * psz, UINT cch);
// free a BSTR
```

---

[13] З Хотя автор и находит строковые процедуры из ustring.h более чем подходящими для управления обработкой текстов в COM, библиотеки ATL и MFC используют несколько иной подход, основанный на alloca и макросах. Более подробную информацию об этих подходах можно прочитать в соответствующей документации.

```
// освобождаем BSTR void SysFreeString(BSTR bstr);
// peek at length-prefix as characters or bytes
// считываем префикс длины как число символов или байт
UINT SysStringLen(BSTR bstr);
UINT SysStringByteLen(BSTR bstr);
```

При пересылке строк методу в качестве параметров типа [in] вызывающий объект должен заботиться о том, чтобы вызвать SysAllocString прежде, чем запускать сам метод, и чтобы вызвать SysFreeString после того, как метод закончил работу. Рассмотрим следующее определение метода:

```
HRESULT SetString([in] BSTR bstr);
```

Пусть в вызывающей программе уже имеется строка, совместимая с OLECHAR, тогда для того, чтобы преобразовать строку в BSTR до вызова метода, необходимо следующее:

```
// convert raw OLECHAR string to a BSTR
// преобразовываем «сырую» строку OLECHAR в строку BSTR
BSTR bstr = SysAllocString(OLESTR(«Hello»));
// invoke method
// вызываем метод HRESULT hr = p->SetString(bstr);
// free BSTR
// освобождаем BSTR SysFreeString(bstr);
```

Промежуточный класс для работы с BSTR, _UBSTR, включен в заголовочный файл ustring.h:

```
// from ustring.h (book-specific header file)
// из ustring.h (специфический для данной книги заголовочный файл)
class _UBSTR

BSTR m_bstr;
public:
_UBSTR(const char *psz) : m_bstr(SysAllocStringLen(0, strlen(psz)))

mbstowcs(m_bstr, psz, INT_MAX);

_UBSTR(const wchar_t *pwsz) : m_bstr(SysAllocString(pwsz))


operator BSTR (void) const
 return m_bstr;
~_UBSTR(void)
 SysFreeString(m_bstr);
;
```

При наличии такого промежуточного класса предыдущий фрагмент кода значительно упростится:

```
// invoke method
// вызываем метод
HRESULT hr = p->SetString(_UBSTR(OLESTR(«Hello»)));
```

Заметим, что в промежуточном классе UBSTR могут быть в равной степени использованы строки типов char и wchar_t.

При передаче из метода строк через параметры типа [out] объект обязан вызвать SysAllocString, чтобы записать результирующую строку в буфер. Затем вызывающий объект должен освободить буфер путем вызова SysFreeString. Рассмотрим следующее определение метода:

```
HRESULT GetString([out, retval] BSTR *pbstr);
```

При реализации метода потребуется создать новую BSTR-строку для возврата вызывающему объекту:

```
STDMETHODIMP MyClass::GetString(BSTR *pbstr)

*pbstr = SysAllocString(OLESTR(«Coodbye!»)) ;
return S_OK;
```

Теперь вызывающий объект должен освободить строку сразу после того, как она скопирована в управляемый приложением строковый буфер:

```
extern OLECHAR g_wsz[];
BSTR bstr = 0;
HRESULT hr = p->GetString(&bstr);
if (SUCCEEDED(hr))

wcscpy(g_wsz, bstr); SysFreeString(bstr);
```

Тут нужно рассмотреть еще один важный аспект BSTR. В качестве BSTR можно передать нулевой указатель, чтобы указать на пустую строку. Это означает, что предыдущий фрагмент кода не совсем корректен. Вызов wcscpy:

```
wcscpy(g_wsz, bstr);
```

должен быть защищен от возможных нулевых указателей:

```
wcscpy (g_wsz, bstr ? bstr : OLESTR(""));
```

Для упрощения использования BSTR в заголовочном файле ustring.h содержится простая встраиваемая функция:

```
intline OLECHAR *SAFEBSTR(BSTR b)

return b ? b : OLESTR("");
```

Разрешение использовать нулевые указатели в качестве BSTR делает тип данных более эффективным с точки зрения использования памяти, хотя и приходится засорять код этими простыми проверками.

Простые типы, показанные на рис. 2.6, могут компоноваться вместе с применением

структур языка C. IDL подчиняется правилам C для пространства имен тегов (tag namespace). Это означает, что большинство IDL-определений интерфейсов либо используют операторы определения типа (typedef):

```
typedef struct tagCOLOR

double red;
double green;
double blue;
 COLOR;
HRESULT SetColor([in] const COLOR *pColor);
```

либо должны использовать ключевое слово struct для квалификации имени тега:

```
struct COLOR  double red; double green; double blue; ;
HRESULT SetColor([in] const struct COLOR *pColor);
```

Первый вариант предпочтительней. Простые структуры, подобные приведенной выше, можно использовать как из Visual Basic, так и из Java. Однако в то время, когда пишется эта книга, текущая версия Visual Basic может обращаться только к интерфейсам, использующим структуры, но она не может быть использована для реализации интерфейсов, в которых структуры являются параметрами методов.

IDL и COM поддерживают также объединения (unions). Для обеспечения однозначной интерпретации объединения IDL требует, чтобы в этом объединении имелся дискриминатор (discriminator), который показывал бы, какой именно член объединения используется в данный момент. Этот дискриминатор должен быть целого типа (integral type) и должен появляться на том же логическом уровне, что и само объединение. Если объединение объявлено вне области действия структуры, то оно считается неинкапсулированным (nonencapsulated):

```
union NUMBER

[case(1)] long i;
[case(2)] float f;
;
```

Атрибут [case] применен для установления соответствия каждого члена объединения своему дискриминатору. Для того чтобы связать дискриминатор с неинкапсулированным объединением, необходимо применить атрибут [switch_is]:

```
HRESULT Add([in, switch_is(t)] union NUMBER *pn, [in] short t);
```

Если объединение заключено вместе со своим дискриминатором в окружающую структуру, то этот составной тип (aggregate type) называется инкапсулированным, или *размеченным* объединением (discriminated union):

```
struct UNUMBER
 short t; [switch_is(t)]
union VALUE

[case(1)] long i;
[case(2)] float f;
```

;
;

COM предписывает для использования с Visual Basic одно общее размеченное объединение. Это объединение называется VARIANT[14] и может хранить объекты или ссылки на подмножество базовых типов, поддерживаемых IDL. Каждому из поддерживаемых типов присвоено соответствующее значение дискриминатора:

VT_EMPTY nothing
VT_NULL SQL style Null
VT_I2 short
VT_I4 long
VT_R4 float
VT_R8 double
VT_CY CY (64-bit currency)
VT_DATE DATE (double)
VT_BSTR BSTR
VT_DISPATCH IDispatch *
VT_ERROR HRESULT
VT_BOOL VARIANT_BOOL (True=-1, False=0)
VT_VARIANT VARIANT *
VT_UNKNOWN IUnknown *
VT_DECIMAL 16 byte fixed point
VT_UI1 opaque byte

Следующие два флага можно использовать в сочетании с вышеприведенными тегами, чтобы указать, что данный вариант (variant) содержит ссылку или массив данного типа:
VT_ARRAY Указывает, что вариант содержит массив SAFEARRAY
VT_BYREF Указывает, что вариант является ссылкой
COM предлагает несколько API-функций для управления VARIANT:

```
// initialize a variant to empty
// обнуляем вариант
void VariantInit(VARIANTARG * pvarg);
// release any resources held in a variant
// освобождаем все ресурсы, используемые в варианте
HRESULT VariantClear(VARIANTARG * pvarg);
// deep-copy one variant to another
// полностью копируем один вариант в другой
HRESULT VariantCopy(VARIANTARG * plhs, VARIANTARG * prhs);
// dereference and deep-copy one variant into another
// разыменовываем и полностью копируем один вариант в другой
HRESULT VariantCopyInd(VARIANT * plhs, VARIANTARG * prhs);
// convert a variant to a designated type
// преобразуем вариант к указанному типу
HRESULT VariantChangeType(VARIANTARG * plhs, VARIANTARG * prhs, USHORT wFlags, VARTYPE vtlhs);
```

[14] 4 К нему можно обращаться и как к VARIANTARG. Термин VARIANTARG относится к вариантам, которые являются допустимыми типами параметров. Термин же VARIANT относится к вариантам, которые являются допустимыми результатами методов. Тип данных VARIANTARG является просто псевдонимом для VARIANT, и оба этих типа могут использоваться равнозначно.

```
// convert a variant to a designated type
// преобразуем вариант к указанному типу (с явным указанием кода локализации)
HRESULT VariantChangeTypeEx(VARIANTARG * plhs, VARIANTARG * prhs, LCID lcid, USHORT wFlags, VARTYPE vtlhs);
```

Эти функции значительно упрощают управление VARIANT'ами. Чтобы понять, как используются эти API-функции, рассмотрим метод, принимающий VARIANT в качестве [in]-параметра:

```
HRESULT UseIt([in] VARIANT var);
```

Следующий фрагмент кода демонстрирует, как передать в этот метод целое число:

```
VARIANT var;
VariantInit(&var);
// initialize VARIANT
// инициализируем VARIANT
V_VT(&var) = VT_I4;
// set discriminator
// устанавливаем дискриминатор
V_I4(&var) = 100;
// set union
// устанавливаем объединение
HRESULT hr = pItf->UseIt(var);
// use VARIANT
// используем VARIANT
VariantClear(&var);
// free any resources in VARIANT
// освобождаем все ресурсы VARIANT
```

Отметим, что этот фрагмент кода использует макросы стандартного аксессора (accessor) для доступа к элементам данных VARIANT. Две следующие строки

```
V_VT(&var) = VT_I4;
V_I4(&var) = 100;
```

эквивалентны коду, который обращается к самим элементам данных:

```
var.vt = VT_I4;
var.lVal = 100;
```

Первый вариант предпочтительнее, так как он может компилироваться на C-трансляторах, которые не поддерживают неименованных объединений.

Ниже приведен пример того, как с помощью приведенной выше технологии реализация метода использует параметр VARIANT в качестве строки:

```
STDMETHODIMP MyClass::UseIt( /*[in] */ VARIANT var)

// declare and init a second VARIANT
// объявляем и инициализируем второй VARIANT
VARIANT var2;
VariantInit(&var2);
```

```
// convert var to a BSTR and store it in var2
// преобразуем переменную в BSTR и заносим ее в var2
HRESULT hr = VariantChangeType(&var2, &var, 0, VT_BSTR);
// use the string
// используем строку
if (SUCCEEDED(hr))

ustrcpy(m_szSomeDataMember, SAFEBSTR(V_BSTR(&var2)));
// free any resources held by var2
// освобождаем все ресурсы, поддерживаемые var2
VariantClear(&var2);

return hr;
```

Отметим, что API-процедура VariantChangeType способна осуществлять сложное преобразование любого переданного клиентом типа из VARIANT в нужный тип (в данном случае BSTR).

Один из последних типов данных, который вызывает дискуссию, – это интерфейс COM. Интерфейсы COM могут быть переданы в качестве параметров метода одним из двух способов. Если тип интерфейсного указателя известен на этапе проектирования, то тип интерфейса может быть объявлен статически:

```
HRESULT GetObject([out] IDog **ppDog);
```

Если же тип на этапе проектирования неизвестен, то разработчик интерфейса может дать пользователю возможность задать тип на этапе выполнения. Для поддержки динамически типизируемых интерфейсов в IDL имеется атрибут [iid_is]:

```
HRESULT GetObject([in] REFIID riid, [out, iid_is(riid)] IUnknown **ppUnk);
```

Хотя эта форма будет работать вполне хорошо, следующий вариант предпочтительнее из-за его подобия с QueryInterface:

```
HRESULT GetObject([in] REFIID riid, [out, iid_is(riid)] void **ppv);
```

Атрибут [iid_is] можно использовать как с параметрами [in], так и [out] для типов IUnknown * или void *. Для того чтобы использовать параметр интерфейса с динамически типизируемым типом, необходимо просто установить IID указателя требуемого типа:

```
IDog *pDog = 0; HRESULT hr = pItf->GetObject(IID_IDog, (void**)&pDog);
```

Соответствующая реализация для инициализации этого параметра просто использовала бы метод QueryInterface для нужного объекта:

```
STDMETHODIMP Class::GetObject(REFIID riid, void **ppv)

extern IUnknown * g_pUnkTheDog;
return g_pUnkTheDog->QueryInterface(riid, ppv);
```

Для уменьшения количества дополнительных вызовов методов между клиентом и

объектом указатели интерфейса с динамически типизируемым типом должны *всегда* использоваться вместо указателей интерфейса со статически типизируемым типом IUnknown.

Атрибуты и свойства

Иногда бывает полезно показать, что объект имеет некие открытые свойства, которые могут быть доступны и/или которые можно модифицировать через COM-интерфейс. COM IDL позволяет аннотировать методы интерфейса с тем, чтобы данный метод либо модифицировал, либо читал именованный атрибут объекта. Рассмотрим такое определение интерфейса:

```
[ object, uuid(0BB3DAE1-11F4-11d1-8C84-0080C73925BA) ]
interface ICollie : IDog

// Age is a read-only property
// Age (возраст) – это свойство только для чтения
[propget] HRESULT Age([out, retval] long *pVal);
// HairCount is a read/write property
// HairCount (счетчик волос) – свойство для чтения/записи
[propget] HRESULT HairCount([out, retval] long *pVal);
[propput] HRESULT HairCount([in] long val);
// CurrentThought is a write-only property
// CurrentThought (текущая мысль) – свойство только для записи
[propput] HRESULT CurrentThought([in] BSTR val);
```

Использование атрибутов *[propget]* и *[propput]* информирует компилятор IDL, что методы, которые ему соответствуют, должны быть отображены в преобразователи свойств (property mutators) или в аксессоры на языках, явно поддерживающих свойства. Применительно к Visual Basic это означает, что элементами *Age* , *HairCount* и *CurrentThought* можно манипулировать, используя тот же синтаксис, как при обращении к элементам структуры:

```
Sub UseCollie(fido as ICollie)
fido.HairCount = fido.HairCount – (fido.Age * 1000)
fido.CurrentThought = «I wish I had a bone»
End Sub
```

C++-отображение этого интерфейса просто прибавляет к именам методов конструкции put или get, чтобы подсказать программисту, что обращение относится к свойству:

```
void UseCollie(ICollie *pFido)

long n1, n2;
HRESULT hr = pFido-&gt;getHairCount(&amp;n1);
assert(SUCCEEDED(hr));
hr = pFido-&gt;getAge(&amp;n2);
assert(SUCCEEDED(hr));
hr = pFido-&gt;putHairCount(n1 – (n2 * 1000)): assert(SUCCEEDED(hr));
BSTR bstr = SysAllocString(OLESTR(«I wish I had a bone»));
```

```
hr = pFido-&gt;putCurrentThought(bstr);
assert(SUCCEEDED(hr));
SysFreeString(bstr);
```

Хотя свойства напрямую не обеспечивают развития, они полезны для выполнения точных преобразований на те языки, которые их поддерживают[15].

Исключения

COM имеет специфическую поддержку выбрасывания (throwing) исключительных ситуаций из реализации методов. Поскольку в языке C++ не существует двоичного стандарта для исключений, COM предлагает явные API-функции для выбрасывания и перехвата объектов COM-исключений:

```
// throw an exception
// возбуждаем исключения
HRESULT SetErrorInfo([in] ULONG reserved, //
m.b.z. [in] IErrorlnfo *pei);
// catch an exception
// перехватываем исключение
HRESULT GetErrorInfo([in] ULONG reserved, // m.b.z.
[out] IErrorInfo **ppei);
```

Процедура SetErrorInfo вызывается из реализации метода, чтобы связать объект исключения с текущим логическим потоком (logical thread)[16]. GetErrorInfo выбирает объект исключения из текущего логического потока и сбрасывает исключение, так что следующие вызовы GetErrorInfo возвратят SFALSE, показывая тем самым, что необработанных исключений нет. Как следует из приведенных ниже подпрограмм, объекты исключений должны поддерживать по крайней мере интерфейс IErrorInfo:

```
[ object, uuid(1CF2B120-547D-101B-8E65-08002B2BD119) ]
interface IErrorInfo: IUnknown

// get IID of interface that threw exception
// получаем IID того интерфейса, который возбудил исключение
HRESULT GetGUID([out] GUID * pGUID);
// get class name of object that threw exception
// получаем имя класса того объекта, который возбудил исключение
HRESULT GetSource([out] BSTR * pBstrSource);
// get human-readable description of exception
// получаем читабельное описание исключения
HRESULT GetDescription([out] BSTR * pBstrDescription);
// get WinHelp filename of documentation of error
```

---

15 1 Пакет Direct-to-COM фирмы Microsoft позволяет клиентам использовать свойства как открытые элементы данных интерфейса с помощью некоего очень хитрого механизма.

16 *1* Спецификация COM использует термин *логический поток (logical thread)* для наименования последовательности вызовов методов, которая может превосходить физический OS-поток.

```
// получаем имя файла WinHelp, содержащего документацию об ошибке
HRESULT GetHelpFile([out] BSTR * pBstrHelpFile);
// get WinHelp context ID for documentation of error
// получаем контекстный идентификатор WinHelp для документации ошибки
HRESULT GetHelpContext([out] DWORD * pdwHelpContext);
```

Специальные объекты исключений могут выбрать другие специфические для исключений интерфейсы в дополнение к *IErrorInfo* .

COM предусматривает по умолчанию реализацию *IErrorInfo* , которую можно создать с использованием API-функции COM *CreateErrorInfo* :

```
HRESULT CreateErrorInfo([out] ICreateErrorInfo **ppcei);
```

В дополнение к *IErrorInfo* объекты исключений по умолчанию открывают интерфейс *ICreateErrorInfo* , чтобы позволить пользователю инициализировать состояние нового исключения:

```
[ object, uuid(22F03340-547D-101B-8E65-08002B2BD119) ]
interface ICreateErrorInfo: IUnknown

// set IID of interface that threw exception
// устанавливаем IID интерфейс, который возбудил исключение
HRESULT SetGUID([in] REFGUID rGUID);
// set class name of object that threw exception
// устанавливаем классовое имя объекта, который возбудил исключение
HRESULT SetSource([in, string] OLECHAR* pwszSource);
// set human-readable description of exception
// устанавливаем читабельное описание исключения
HRESULT SetDescription([in, string] OLECHAR* pwszDesc);
// set WinHelp filename of documentation of error
// устанавливаем имя файла WinHelp, содержащего документацию об ошибке
HRESULT SetHelpFile([in, string] OLECHAR* pwszHelpFile);
// set WinHelp context ID for documentation of error
// устанавливаем идентификатор контекста WinHelp для документации ошибки
HRESULT SetHelpContext([in] DWORD dwHelpContext);
```

Заметим, что этот интерфейс просто позволяет пользователю заполнить объект исключения пятью основными атрибутами, доступными из интерфейса *IErrorInfo* .

Следующая реализация метода выбрасывает COM-исключение своему вызывающему объекту, используя объект исключений по умолчанию:

```
STDMETHODIMP PugCat::Snore(void)

if (this->IsAsleep())
// ok to perform operation?
// можно ли выполнять операцию?
return this->DoSnore();
//do operation and return
// выполняем операцию и возвращаемся
//otherwise create an exception object
```

```
// в противном случае создаем объект исключений
ICreateErrorInfo *pcei = 0; HRESULT hr = CreateErrorInfo(&pcei);
assert(SUCCEEDED(hr));
// initialize the exception object
// инициализируем объект исключений
hr = pcei->SetGUID(IIDIPug);
assert(SUCCEEDED(hr));
hr = pcei->SetSource(OLESTR(«PugCat»));
assert(SUCCEEDED(hr));
hr = pcei->SetDescription(OLESTR("I am not asleep !"));
assert(SUCCEEDED(hr));
hr = pcei->SetHelpFile(OLESTR(«C:\PugCat.hlp»));
assert(SUCCEEDED(hr));
hr = pcei->SetHelpContext(5221);
assert(SUCCEEDED(hr));
// «throw» exception
// «выбрасываем» исключение
IErrorInfo *pei = 0;
hr = pcei->QueryInterface(IIDIErrorInfo , (void**)&pei);
assert(SUCCEEDED(hr));
hr = SetErrorInfo(0, pei);
// release resources and return a SEVERITYERROR result
// высвобождаем ресурсы и возвращаем результат
// SEVERITY ERROR (серьезность ошибки)
pei->Release();
pcei->Release();
return PUGEPUGNOTASLEEP;
```

Отметим, что поскольку объект исключений передается в процедуру *SetErrorInfo* , COM сохраняет ссылку на исключение до тех пор, пока оно не будет «перехвачено» вызывающим объектом, использующим *GetErrorInfo* .

Объекты, которые сбрасывают исключения COM, должны использовать интерфейс *ISupportErrorInfo* , чтобы показывать, какие интерфейсы поддерживают исключения. Этот интерфейс используется клиентами, чтобы установить, верный ли результат дает *GetErrorInfo* [17]. Этот интерфейс предельно прост:

```
[ object, uuid(DFOB3060-548F-101B-8E65-08002B2BD119) ]
interface ISupportErrorInfo: IUnknown

HRESULT InterfaceSupportsErrorInfo([in] REFIID riid);
```

Предположим, что класс *PugCat* , рассмотренный в этой главе, сбрасывает исключения из каждого поддерживаемого им интерфейса. Тогда его реализация будет выглядеть так:

```
STDMETHODIMP PugCat::InterfaceSupportsErrorInfo(REFIID riid)
```

---

[17] *2* Объект, который обеспечивает выполнение *GetErrorInfo* , декларирует, что он явно программирует с использованием исключений COM и что никаких ошибочных исключений, сброшенных объектами младшего уровня, случайно не распространилось.

if (riid == II*DIAnimal* || *riid == IID* ICat || riid == II*DIDog* || *riid == IID* IPug) return *SOK; else return S* FALSE;

Ниже приведен пример клиента, который надежно обрабатывает исключения, используя *ISupportErrorInfo* и *GetErrorInfo* :

```
void TellPugToSnore(/*[in]*/ IPug *pPug)

// call a method
// вызываем метод
HRESULT hr = pPug->Snore();
if (FAILED(hr))

// check to see if object supports COM exceptions
// проверяем, поддерживает ли объект исключения COM
ISupportErrorInfo *psei = 0;
HRESULT hr2 =pPug->QueryInterface( IIDISupportErrorInfo, (void**)&psei);
if (SUCCEEDED(hr2))

// check if object supports COM exceptions via IPug methods
// проверяем, поддерживает ли объект исключения COM через методы
IPug hr2 = psei->InterfaceSupportsErrorInfo(IIDIPug);
if (hr2 == SOK)

// read exception object for this logical thread
// читаем объект исключений для этого логического потока
IErrorInfo *pei = 0;
hr2 = GetErrorInfo(0, &pei);
if (hr2 == SOK)

// scrape out source and description strings
// извлекаем строки кода и описания
BSTR bstrSource = 0, bstrDesc = 0;
hr2 = pei->GetDescription(&bstrDesc);
assert(SUCCEEDED(hr2));
hr2 = pei->GetSource(&bstrSource);
assert(SUCCEEDED(hr2));
// display error information to end-user
// показываем информацию об ошибке конечному пользователю
MessageBoxW(0, bstrDesc ? bstrDesc : L"«, bstrSource ? bstrSource : L»", MBOK);
// free resources
// высвобождаем ресурсы
SysFreeString(bstrSource);
SysFreeString(bstrDesc);
pei->Release();


psei->Release();


if (hr2!= SOK)
```

```
// something went wrong with exception
// что-то неладно с исключением
MessageBox(0, «Snore Failed», «IPug», MBOK);
```

Довольно просто отобразить исключения COM в исключения C++, причем в любом месте. Определим следующий класс, который упаковывает объект исключений COM и *HRESULT* в один класс C++:

```
struct COMException

HRESULT mhresult;
// hresult to return
// hresult для возврата IErrorInfo *mpei;
// exception to throw
// исключение для выбрасывания
COMException(HRESULT hresult, REFIID riid, const OLECHAR *pszSource, const
OLECHAR *pszDesc, const OLECHAR *pszHelpFile = 0, DWORD dwHelpContext = 0)

// create and init an error info object
// создаем и инициализируем объект информации об ошибке
ICreateErrorInfo *pcei = 0;
HRESULT hr = CreateErrorInfo(&pcei);
assert(SUCCEEDED(hr));
hr = pcei->SetGUID(riid);
assert(SUCCEEDED(hr));
if (pszSource) hr=pcei->SetSource(constcast<OLECHAR*>(pszSource));
assert(SUCCEEDED(hr));
if (pszDesc) hr=pcei->SetDescription(constcast<OLECHAR*>(pszDesc));
assert(SUCCEEDED(hr));
if (pszHelpFile) hr=pcei->SetHelpFile(constcast<OLECHAR*>(pszHelpFile));
assert(SUCCEEDED(hr));
hr = pcei->SetHelpContext(dwHelpContext);
assert(SUCCEEDED(hr));
// hold the HRESULT and IErrorInfo ptr as data members
// сохраняем HRESULT и указатель IErrorInfo как элементы данных
mhresult = hresult;
hr=pcei->QueryInterface(IIDIErrorInfo, (void**)&mpei);
assert(SUCCEEDED(hr));
pcei->Release();

;
```

С использованием только что приведенного C++-класса *COMException* показанный выше метод *Snore* может быть модифицирован так, чтобы он преобразовывал любые исключения C++ в исключения COM:

```
STDMETHODIMP PugCat::Snore(void)

HRESULT hrex = SOK;
try
```

```
if (this->IsAsleep()) return this->DoSnore();
else throw COMException(PUGEPUGNOTASLEEP, IIDIPug, OLESTR(«PugCat»), OLESTR(«I am not asleep!»));

catch (COMException& ce)

// a C++ COMException was thrown
// было сброшено исключение COMException C++
HRESULT hr = SetErrorInfo(0, ce.mpei);
assert(SUCCEEDED(hr));
ce.mpei->Release();
hrex = ce.mhresult;

catch (…)

// some unidentified C++ exception was thrown
// было выброшено какое-то неидентифицированное исключение C++
COMException ex(EFAIL, IIDIPug, OLESTR(«PugCat»), OLESTR(«A C++ exception was thrown»));
HRESULT hr = SetErrorInfo(0, ex.mpei);
assert(SUCCEEDED(hr));
ex.mpei->Release();
hrex = ex.mhresult;

return hrex;
```

Заметим, что реализация метода заботится о том, чтобы не позволить чисто C++-исключениям переходить через границы метода. Таково безусловное требование COM.

Где мы находимся?

В этой главе была представлена концепция интерфейса COM. Интерфейсы COM обладают простыми двоичными сигнатурами, которые позволяют любому клиенту обращаться к объекту независимо от языка программирования, использованного клиентом или конструктором объекта. Чтобы облегчить поддержку различных языков, интерфейсы COM определяются на языке *IDL* (Interface Definition Language). Эти IDL-определения интерфейса могут быть также использованы для генерирования кода передачи данных (communications code), который позволяет получать доступ к объекту через сеть.

Большая часть этой главы была посвящена *IUnknown* – базовому интерфейсу, на котором построен весь COM. Все интерфейсы COM должны наследовать от *IUnknown* . Следовательно, все объекты COM должны реализовывать IUnknown. В *IUnknown* предусмотрено три сигнатуры метода, посредством которых клиент может безошибочно управлять иерархией типов объекта для доступа к дополнительным возможностям, предоставляемым этим объектом. С учетом этого *QueryInterface* можно рассматривать как оператор приведения типа в COM. По этой же причине *IUnknown* можно рассматривать как *«void *»* (указатель на пустой тип) среди указателей интерфейса, так как от него не слишком много пользы до тех пор, пока он не «приведен» (is «cast») к чему-нибудь более содержательному с помощью *QueryInterface* .

Следует заметить, что при обращении или реализации *IUnknown* не было сделано никаких существенных системных вызовов. В этом смысле *IUnknown* просто является

протоколом или набором обещаний (promises), которого должны придерживаться все программы. Это позволяет объектам COM быть очень простыми и эффективными. Реализация *IUnknown* в C++ требует всего нескольких строк стандартного кода. Чтобы автоматизировать реализацию *IUnknown* в C++, была представлена серия макросов для препроцессора, которые реализуют *QueryInterface* под табличным управлением. Хотя эти макросы не были совершенно необходимыми, они удаляли большую часть общего стандартного кода из каждого определения класса, не внося при этом заметных усложнений в реализацию.

Глава 3. Классы

int cGorillas = Gorilla::GetCount();
*IApe *pApe = new Gorilla();*
pApe->GetYourStinkingPawsOffMeYouDamnDirtyApe();
Charleton Heston, 1968

В предыдущей главе обсуждались принципы интерфейсов COM вообще и интерфейс *IUnknown* в частности. Были показаны способы управления указателями интерфейса из C++, и детально обсуждалась фактическая техника реализации *IUnknown* . Однако не обсуждалось, как обычно клиенты получают начальный указатель интерфейса на объект, или как средства реализации объекта допускают, чтобы их объекты могли быть обнаружены внешними клиентами. В данной главе демонстрируется, как реализации объектов COM интегрируют в среду выполнения COM, чтобы дать клиентам возможность найти или создать объекты требуемого конкретного типа.

Снова об интерфейсе и реализации

В предыдущей главе интерфейс COM был определен как абстрактный набор операций, выражающий некоторую функциональность, которую может экспортировать объект. Интерфейсы COM описаны на языке *IDL* (Interface Definition Language – язык определений интерфейса) и имеют логические имена, которые указывают на моделируемые ими функциональные возможности. Ниже приведено IDL-определение COM-интерфейса *IApe* :

```
[object, uuid(753A8A7C-A7FF-11d0-8C30-0080C73925BA)]
interface IApe : Unknown

import «unknwn.idl»;
HRESULT EatBanana(void);
HRESULT SwingFromTree(void);
[propget] HRESULT Weight([out, retval] long *plbs);
```

Сопровождающая *IApe* документация должна специфицировать примерную семантику трех операций: *EatBanana* , *SwingFromTree* и *Weight* . Все объекты, раскрывающие *IApe* посредством *QueryInterface* , должны гарантировать, что их реализации этих методов придерживаются семантического контракта *IApe* . В то же время определения интерфейса почти всегда специально оставляют место для интерпретации разработчиком объекта. Это означает, что клиенты никогда не могут быть полностью уверены в точном поведении любого заданного метода, а только в том, что его поведение будет следовать схематическим

правилам, описанным в документации к интерфейсу. Эта контролируемая степень неопределенности является фундаментальной характеристикой полиморфизма и одной из основ развития объектно-ориентированного программного обеспечения.

Рассмотрим только что приведенный интерфейс *IApe* . Вероятно (и даже возможно), что будет более одной реализации интерфейса *IApe* . Поскольку определение *IApe* является общим для всех реализаций, то предположения, которые могут сделать клиенты о поведении метода *EatBanana* , должны быть достаточно неопределенными, чтобы позволить каждой обезьяне – гориллам, шимпанзе и орангутангам (все они могут реализовывать интерфейс *IApe*), получить свои допустимые (но слегка различные) интерпретации данной операции. Без этой гибкости полиморфизм невозможен.

COM определенно трактует интерфейсы, реализации и классы как три различных понятия. Интерфейсы являются абстрактными протоколами для связи с объектом. Реализации – это конкретные типы данных, поддерживающие один или несколько интерфейсов с помощью точных семантических интерпретаций каждой из абстрактных операций интерфейса. Классы – это именованные реализации, представляющие собой конкретные типы, которым можно приписывать значения, и формально называются COM-классами, или *коклассами (coclasses).*

В смысле инкапсуляции о COM-классе известно только его имя и потенциальный список интерфейсов, которые он выставляет. Подобно COM-интерфейсам, COM-классы именуются с использованием *GUID* (globally unique identifier – глобально уникальный идентификатор), хотя если *GUID* используются для именования COM-классов, то они называются идентификаторами класса – *CLSID* . Аналогично именам интерфейсов, эти имена классов должны быть хорошо известны клиенту до того, как он их использует. Поскольку для обеспечения полиморфизма COM-интерфейсы являются семантически неопределенными, то COM не позволяет клиентам просто запрашивать *любую доступную* реализацию данного интерфейса. Вместо этого клиенты должны точно специфицировать требуемую реализацию. Это лишний раз подчеркивает тот факт, что COM-интерфейсы – это всего лишь абстрактные коммуникационные протоколы, единственное назначение которых – обеспечить клиентам связь с объектами, принадлежащими конкретным, имеющим ясную цель классам реализации [18].

Кроме того, что реализации могут быть именованы с помощью *CLSID* , COM поддерживает текстовые псевдонимы, так называемые *программные идентификаторы (programmatic identifiers),* иначе *ProgID.* Эти *ProgID* поступают в формате *libraryname.classname.version* и, в отличие от CLSID, являются уникальными только по соглашению. Клиенты могут преобразовывать *ProgID* в *CLSID* и обратно с помощью API-функций COM *CLSIDFromProgID* и *ProgIDFromCLSID* :

```
HRESULT CLSIDFromProgID([in, string] const OLECHAR *pwszProgID, [out] CLSID
*pclsid);
HRESULT ProgIDFromCLSID([in] REFCLSID rclsid, [out, string] OLECHAR
**ppwszProgID);
```

Для преобразования *ProgID* в *CLSID* нужно просто вызвать *CLSIDFromProgID* :

---

[18] *1* Хотя и мало смысла запрашивать «любую доступную реализацию» данного интерфейса, иногда имеет смысл произвести семантическое группирование реализаций, имеющих определенные общие черты высокого уровня, например, чтобы все они были животными или чтобы все они имели службу регистрации. Чтобы обеспечить обнаружение этого типа компонентов, COM поддерживает объявление такой систематики (taxonomy) посредством использования *категорий компонентов (component categories).* Поскольку часто это тот случай, когда все классы, принадлежащие к одной категории компонентов, будут реализовывать одно и то же множество интерфейсов, то такое условие, без сомнения, является достаточным для принадлежности к одной категории компонентов.

```
HRESULT GetGorillaCLSID(CLSID& rclsid)

const OLECHAR wszProgID[] = OLESTR(«Apes.Gorilla.1»);
return CLSIDFromProgID(wszProgID, &rclsid);
```

На этапе выполнения будет просматриваться база данных конфигураций COM для преобразования *ProgID Apes.Gorilla.1* в *CLSID* , соответствующий классу реализации COM.

Объекты классов

Основное требование всех COM-классов состоит в том, что они должны иметь объект класса. Объект класса – это единственный экземпляр (синглетон), связанный с каждым классом, который реализует функциональность класса, общую для всех его экземпляров. Объект класса ведет себя как метакласс по отношению к заданной реализации, а реализуемые им методы выполняют роль статических функций-членов из C++. По логике вещей, может быть только один объект класса в каждом классе; однако в силу распределенной природы COM каждый класс может иметь по одному объекту класса на каждую хост-машину (host machine), на учетную запись пользователя или на процесс, – в зависимости от того, как используется этот класс. Первой точкой входа в реализацию класса является ее объект класса.

Объекты класса являются очень полезными программистскими абстракциями. Объекты класса могут вести себя как известные объекты (когда их идентификатор CLSID выступает в качестве имени объекта), которые позволяют нескольким клиентам связываться с одним и тем же объектом, определенным с помощью данного CLSID. В то время как системы в целом могли быть созданы с использованием исключительно объектов класса, объекты класса часто используются как посредники (brokers) при создании новых экземпляров класса или для того, чтобы найти имеющиеся экземпляры, определенные с помощью какого-нибудь известного имени объекта. При использовании в этой роли объект класса обычно объявляет только один или два промежуточных интерфейса, которые позволят клиентам создать или найти те экземпляры, которые в конечном счете будут выполнять нужную работу. Например, рассмотрим описанный ранее интерфейс *IApe* . Объявление интерфейса *IApe* не нарушит законы COM для объекта класса:

```
class GorillaClass : public IApe

public:
// class objects are singletons, so don't delete
// объекты класса существуют в единственном экземпляре,
// так что не удаляйте их
IMPLEMEN*TUNKNOWN* N*ODELETE (GorillaClass)*
*BEGIN* INTERFAC*ETABLE(GorillaClass)*
*IMPLEMENTS* INTERFACE(IApe)
EN*DINTERFACE* TABLE()
// IApe methods
// методы IApe
STDMETHODIMP EatBanana(void);
STDMETHODIMP SwingFromTree(void);
STDMETHODIMP ge*tWeight(long *plbs);*
;
```

Если для данного класса C++ может существовать лишь один экземпляр (так ведут себя все объекты классов в COM), то в любом заданном экземпляре может быть только одна горилла (gorilla). Для некоторых областей одноэлементных множеств достаточно. В случае с гориллами, однако, весьма вероятно, что клиенты могут захотеть создавать приложения, которые будут использовать несколько *различных* горилл одновременно. Чтобы обеспечить такое использование, объект класса не должен экспортировать интерфейс ***IApe*** , а вместо этого должен экспортировать новый интерфейс, который позволит клиентам создавать новых горилл и/или находить известных горилл по их имени. Это потребует от разработчика определить два класса C++: один для реализации объекта класса и другой для реализации действительных экземпляров класса. Для реализации гориллы класс C++, который определяет экземпляры гориллы, будет реализовывать интерфейс *IApe* :

```
class Gorilla : public IApe

public:
// Instances are heap-based, so delete when done
// копии размещены в куче, поэтому удаляем после выполнения
IMPLEMEN*TUNKNOWN()*
*BEGIN* INTERFAC*ETABLE()*
*IMPLEMENTS* INTERFACE(IApe)
EN*DINTERFACE* TABLE()
// IApe methods
// методы IApe
STDMETHODIMP EatBanana(void);
STDMETHODIMP SwingFromTree(void);
STDMETHODIMP ge*tWeight(long *plbs)* :
;
```

Второй интерфейс понадобится для определения тех операций, которые будет реализовывать объект класса *Gorilla* :

```
[object, uuid(753A8AAC-A7FF-11d0-8C30-0080C73925BA)]
interface IApeClass : IUnknown

HRESULT CreateApe([out, retval] IApe **ppApe);
HRESULT GetApe([in] long nApeID, [out, retval] IApe **ppApe);
[propget]
HRESULT AverageWeight([out, retval] long *plbs);
```

Получив это определение интерфейса, объект класса будет реализовывать методы *IApeClass* или путем создания новых экземпляров C++-класса *Gorilla* (в случае *CreateApe*), или преобразованием произвольно выбранного имени объекта (в данном случае типа *integer*) в отдельный экземпляр (в случае *GetApe*):

```
class GorillaClass : public IApeClass

public: IMPLEMEN*TUNKNOWN* N*ODELETE(GorillaClass)*
*BEGIN* INTERFAC*ETABLE(GorillaClass)*
*IMPLEMENTS* INTERFACE(IApeClass)
EN*DINTERFACE* TABLE()
```

```
STDMETHODIMP CreateApe(Ape **ppApe)

if ((*ppApe = new Gorilla) == 0) return EOUTOFMEMORY;
(*ppApe)->AddRef();
return S OK;

STDMETHODIMP GetApe(long nApeID, IApe **ppApe)

// assume that a table of well-known gorillas is
// being maintained somewhere else
// допустим, что таблица для известных горилл
// поддерживается где-нибудь еще
extern Gorilla *grgWellKnownGorillas[];
extern int g nMaxGorillas;
// assert that nApeID is a valid index
// объявляем, что nApeID – допустимый индекс
*ppApe = 0;
if (nApeID > gnMaxGorillas || nApeID < 0) return E INVALIDARG;
// assume that the ID is simply the index into the table
// допустим, что ID – просто индекс в таблице
if ((*ppApe = grgWellKnownGorillas[nApeID]) == 0) return E INVALIDARG;
(*ppApe)->AddRef();
return SOK;

STDMETHODIMP getAverageWeight(long *plbs)

extern *g rgWellKnownGorillas[];
extern int gnMaxGorillas;
*plbs = 0;
long lbs;
for (int i = 0; i < g nMaxGorillas; i++)

grgWellKnownGorillas[i]->get Weight(&lbs);
*plbs += lbs;

// assumes gnMaxGorillas is non-zero
// предполагается , что gnMaxGorillas ненулевой
*plbs /= g nMaxGorillas;
return SOK;

;
```

Отметим, что в этом коде предполагается, что внешняя таблица известных горилл уже поддерживается – или самими копиями *Gorilla* , или каким-нибудь другим посредником (agent).

Активация

Клиентам требуется механизм для поиска объектов класса. В силу динамической природы COM это может привести к загрузке библиотеки DLL или запуску обслуживающего процесса (server process). Эта процедура вызова объекта к жизни называется активацией

объекта.

В COM имеется три модели активации, которые можно использовать для занесения объектов в память, чтобы сделать возможными вызовы методов. Клиенты могут попросить COM связать объект класса с данным классом. Кроме того, клиенты могут попросить, чтобы COM создала новые экземпляры классов, определенные с помощью *CLSID* . Наконец, клиенты могут попросить COM вызвать к жизни перманентный (persistent) объект, состояние которого определено как постоянное. Из этих трех моделей только первая (связывание с объектом класса) является абсолютно необходимой. Две другие модели являются просто оптимизациями обычно применяющихся способов активации. Дополнительные, определенные пользователем, модели активации могут быть реализованы в терминах одного (или более) из этих трех примитивов.

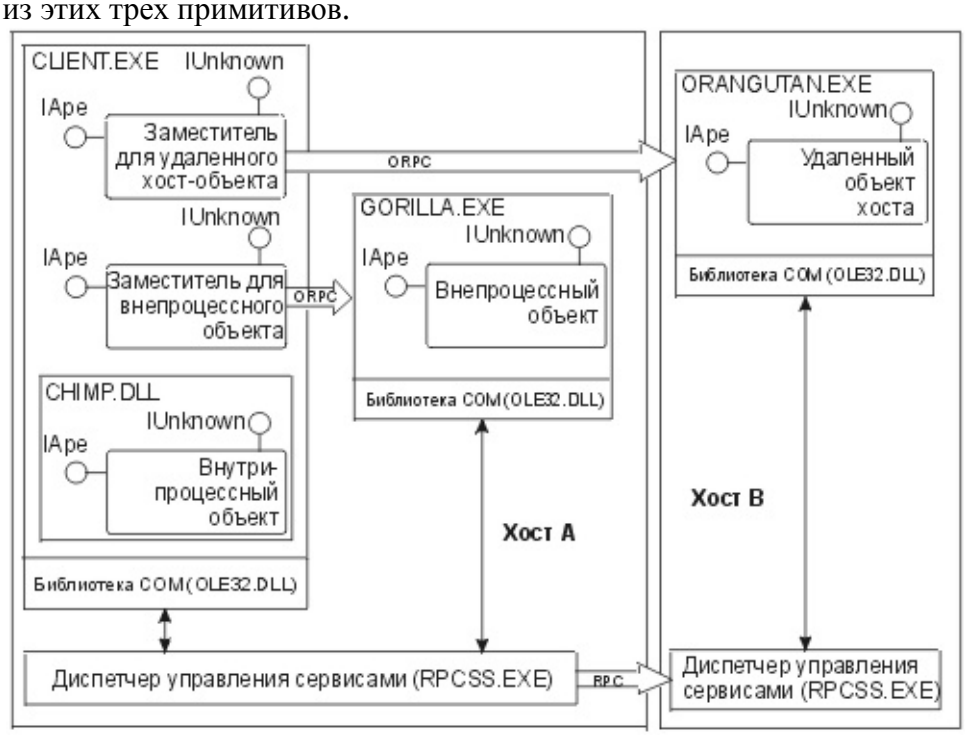

Рис. 3.1. Библиотека COM и SCM

Каждая из описанных трех моделей активации пользуется услугами имеющегося в COM диспетчера управления сервисами *SCM (Service Control Manager*)[19]. *SCM* является основной точкой рандеву для всех запросов на активацию в каждой отдельной машине. Каждая хост-машина, поддерживающая COM, имеет свой собственный локальный SCM, который переадресовывает удаленные запросы на активацию на SCM удаленной машины, где этот запрос будет трактоваться как локальный запрос на активацию. SCM используется только для того, чтобы активировать объект и привязать к нему начальный указатель интерфейса. Как только объект активирован, SCM более не связан с вызовом методов клиента и объекта. Как показано на рис. 3.1, под Windows NT SCM реализован в службе *RPCSS* (Remote Procedure Call Service System – система сервиса удаленного вызова процедур). Службы SCM объявляются в программы как высокоуровневые типы моникеров[20] и как низкоуровневые API-функции, причем все они реализованы в библиотеке COM (как это называется в Спецификации COM). Под Windows NT большая часть библиотеки COM

---

19 *1* В Windows NT также имеется подсистема, известная как Service Control Manager, или просто Services, которая используется для запуска процессов, не зависящих от входа в систему. Далее в этой книге мы будем называть этот диспетчер управления сервисами NT SCM, чтобы отличать его от COM SCM.

20 *2* Моникеры являются поисковыми объектами, которые скрывают детали активации или связывающего алгоритма. Более подробно моникеры обсуждаются далее в этой главе.

реализована в *OLE32.DLL* . Для повышения эффективности библиотека COM может использовать локальный или кэшированный режим, чтобы избежать ненужных запросов службы *RPCSS* со стороны *IPC* (interprocess communication – межпроцессное взаимодействие).

Напомним, что главным принципом COM является разделение интерфейса и реализации. Одной из деталей реализации, скрытых от клиента, является местонахождение реализации объекта. Невозможно определить, не только на каком хосте был активирован объект, но и был ли локальный объект активирован в клиентском процессе или в отдельном процессе на локальной машине. Это дает разработчикам объектов очень большую гибкость при решении того, как и где использовать реализации объектов, учитывая такие проблемы, как устойчивость к сбоям (robustness), обеспечение безопасности, распределение нагрузки и производительность. Клиент имеет возможность во время активации указать свои предпочтения относительно того, где будет активирован объект. Многие клиенты, однако, выражают свое безразличие к данному вопросу. В таком случае этот выбор сделает SCM, исходя из текущей конфигурации нужного класса.

Когда объект активирован внутри процесса, то в процесс клиента загружается та библиотека DLL, которая реализует методы объекта, и все данные-члены хранятся в адресном пространстве клиента. Так как не требуется никаких переключении процессов, то эффективность вызова методов чрезвычайно высока. Кроме того, клиентский поток может быть использован для прямого выполнения кода метода, при условии, что требования по организации поточной обработки (*threading requirements*) объекта соответствуют клиентским требованиям. Если у клиента и у объекта требования по организации поточной обработки совместимы, то также не нужно никаких переключении потоков. Если вызовы метода могут выполняться с использованием клиентского потока, после активации объекта не требуется участия никакой промежуточной среды времени выполнения, и цена вызова метода просто равна вызову виртуальной функции. Это обстоятельство делает COM, встроенный в процесс, особенно хорошо приспособленным для приложений, чувствительных к эффективности выполнения, так как вызов метода обходится не дороже, чем обычный вызов глобальной функции в DLL[21].

Когда объект активирован извне процесса (то есть в другом процессе на локальной или удаленной машине), то код, реализующий методы объекта, выполняется в процессе определенного сервера и все данные-члены объекта сохраняются в адресном пространстве процесса сервера. Чтобы позволить клиенту связываться с внепроцессным (*out-of-process*) объектом, COM прозрачно (скрытно от клиента) возвращает ему «заместитель» (*proxy*) во время активации. В главе 5 подробно обсуждается, что этот «заместитель» выполняется в клиентском потоке и переводит вызовы метода, преобразованные в RPC-запросы (Remote Procedure Call – удаленный вызов процедуры), в контекст исполнения сервера, где эти RPC-запросы затем преобразуются обратно в вызовы метода текущего объекта. Это делает вызов метода менее эффективным, так как при каждом обращении к объекту требуются переключение потока и переключение процесса. К преимуществам внепроцессной (то есть работающей не в клиентском процессе) активации относятся изоляция ошибок, распределение и повышенная безопасность. В главе 6 внепроцессная активация будет рассматриваться подробно.

Использование SCM

Напомним, что SCM поддерживает три примитива активации (связывание с объектами

---

21 *З* Степень изоляции, необходимая для вызова во внешнюю DLL, приблизительно эквивалентна вызову функции через вход таблицы *vtbl* .

класса, связывание с экземплярами класса, связывание с постоянными экземплярами из файлов). Как показано на рис. 3.2, эти примитивы логически разделены на уровни[22]. Примитивом нижнего уровня является связывание с объектом класса. Этот примитив также наиболее прост для понимания.

Вместо того чтобы вручную загружать код класса, клиенты пользуются услугами SCM посредством низкоуровневой API-функции COM *CoGetClassObject* . Эта функция запрашивает SCM присвоить значение указателю на требуемый объект класса:

```
HRESULT CoGetClassObject(
[in] REFCLSID rclsid,
// which class object?
// Какой объект класса?
[in] DWORD dwClsCtx,
// locality?
//местонахождение?
[in] COSERVERINFO *pcsi,
// host/security info
//сведения о сервере и обеспечении безопасности
[in] REFIID riid,
// which interface?
// какой интерфейс?
[out, ii dis(riid)] void **ppv);
// put it here !
// поместим его здесь!
```

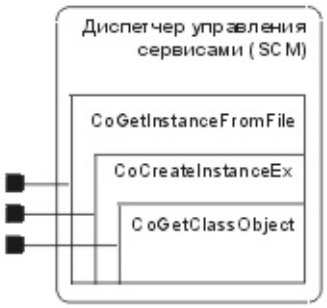

Рис. 3.2. Примитивы активации

Первый параметр в *CoGetClassObject* показывает, какой класс реализации запрашивается. Последний параметр – это ссылка на указатель интерфейса, с которым нужно связаться, а четвертый параметр – это просто *IID* , описывающий тип указателя интерфейса, на который ссылается последний параметр. Более интересные параметры – второй и третий, которые определяют, когда объект класса должен быть активирован.

В качестве второго параметра *CoGetClassObject* принимает битовую маску (bitmask), которая позволяет клиенту указать характеристики скрытого и живучего состояний объекта (например, будет ли объект запущен в процессе, вне процесса или вообще на другом сервере). Допустимые значения для этой битовой маски определены в стандартном перечислении *CLSCTX* :

```
enum tagCLSCTX  CLSCTXINPROC SERVER = 0x1,
// run -inprocess
// запуск в процесс
CLSCTXINPROC HANDLER = 0x2,
```

---

22 *1* Это разделение в значительной степени концептуально, так как библиотека COM и протокол передачи (wire-protocol) реализуют каждый примитив как отдельную ветвь программы и формат пакета.

```
// see note[23]
// смотрите сноску[24]
CLSCTX_LOCAL_SERVER = 0x4,
// run out-of-process
// запуск вне процесса
CLSCTX_REMOTE_SERVER = 0x10
// run off-host
// запуск вне хост-машины
 CLSCTX;
```

Эти флаги могут быть подвергнуты побитному логическому сложению (bit-wise-ORed together), и в случае, когда доступен более чем один запрошенный *CLSCTX* , COM выберет наиболее эффективный тип сервера (это означает, что COM будет, когда это возможно, использовать наименее значимый бит битовой маски). Заголовочные файлы SDK также включают в себя несколько сокращенных макросов, которые сочетают несколько флагов *CLSCTX* , используемых во многих обычных сценариях:

```
#define CLSCTX_INPROC (CLSCTX_INPROC_SERVER |
CLSCTX_INPROC_HANDLER)
#define CLSCTX_SERVER (CLSCTX_INPROC_SERVER |
CLSCTX_LOCAL_SERVER |
CLSCTX_REMOTE_SERVER)
#define CLSCTX_ALL (CLSCTX_INPROC_SERVER |
CLSCTX_INPROC_HANDLER |
CLSCTX_LOCAL_SERVER |
CLSCTX_REMOTE_SERVER)
```

Заметим, что такие среды, как Visual Basic и Java, всегда используют CLSCTX_ALL , показывая тем самым, что подойдет любая доступная реализация.

Третий параметр *CoGetClassObject* – это указатель на структуру, содержащую информацию об удаленном доступе и безопасности. Эта структура имеет тип *COSERVERINFO* и позволяет клиентам явно указывать, какой машине следует активировать объект, а также как конфигурировать установки обеспечения безопасности, используемые при создании запроса на активацию объекта:

---

[23] *2* Внутрипроцессные обработчики (in-process handlers) – в значительной степени пережитки документации OLE. Эти обработчики являются виутрипроцессными компонентами, выступающими в качестве представителей клиентской стороны объекта, который в действительности находится в другом процессе. Обработчики используются в документах OLE для кэширования изображений у клиента с целью сократить поток IPC (interprocess communication – межпроцессное взаимодействие) при перерисовке экрана. Хотя эти обработчики в общем случае производят считывание, они редко используются вне контекста документов OLE. Windows NT 5.0 будет обеспечивать дополнительные возможности для реализации обработчиков, но подробности того, как это будет достигнуто, были еще схематичны во время написания этой книги.

[24] *2* Внутрипроцессные обработчики (in-process handlers) – в значительной степени пережитки документации OLE. Эти обработчики являются виутрипроцессными компонентами, выступающими в качестве представителей клиентской стороны объекта, который в действительности находится в другом процессе. Обработчики используются в документах OLE для кэширования изображений у клиента с целью сократить поток IPC (interprocess communication – межпроцессное взаимодействие) при перерисовке экрана. Хотя эти обработчики в общем случае производят считывание, они редко используются вне контекста документов OLE. Windows NT 5.0 будет обеспечивать дополнительные возможности для реализации обработчиков, но подробности того, как это будет достигнуто, были еще схематичны во время написания этой книги.

```
typedef struct COSERVERINFO

DWORD dwReserved1;
// reserved , must be zero
// зарезервировано, должен быть нуль
LPWSTR pwszName;
// desired host name, or null
// желаемое имя хост-машины или нуль
COAUTHINFO *pAuthInfo;
// desired security settings
// желаемые установки безопасности DWORD dwReserved2;
// reserved, must be zero
// зарезервировано, должен быть нуль
 COSERVERINFO;
```

Если клиент не указывает имя хоста (host name), а использует только флаг CLSCTXREMOTESERVER, то для определения того, какая машина будет активировать объект, COM использует информацию о конфигурации каждого CLSID. Если клиент передает явное имя хоста, то оно получит приоритет перед любыми ранее сконфигурированными именами хостов, о которых может знать COM. Если клиент не желает передавать явную информацию о безопасности или имя хоста в CoGetClassObject, можно применить нулевой указатель COSERVERINFO.

Имея в наличии *CoGetClassObject* , клиент может дать запрос SCM на связывание указателя интерфейса с объектом класса:

```
HRESULT GetGorillaClass(IApeClass * &rpgc)

// declare the CLSID for Gorilla as a GUID
// определяем CLSID для Gorilla как GUID
const CLSID CLSIDGorilla =
 0x571F1680 , 0xCC83, 0x11d0,
 0x8C, 0x48, 0x00, 0x80, 0xC7, 0x39, 0x25, 0xBA
 ;
// call CoGetClassObject directly
// вызываем прямо CoGetClassObject
return CoGetClassObject(CLSIDGorilla , CLSCTXALL , 0, IIDIApeClass , (void**)&rpgc);
```

Отметим, что если запрошенный класс доступен как внутрипроцессный сервер, то COM автоматически загрузит соответствующую DLL и вызовет известную экспортируемую функцию, которая возвращает указатель на требуемый объект класса[25]. Когда вызов *CoGetClassObject* завершен, библиотека COM и SCM полностью выходят из игры. Если бы класс был доступен только с внепроцессного или удаленного сервера, COM вместо этого возвратила бы заместитель, который позволил бы клиенту получить удаленный доступ к объекту класса.

Напомним, что интерфейс *IApeClass* придуман для того, чтобы позволить клиентам находить или создавать экземпляры заданного класса. Рассмотрим следующий пример:

---

[25] *З* Требования параллелизма для класса должны технически соответствовать таким же требованиям в потоке вызова.

```
HRESULT FindAGorillaAndEatBanana(long nGorillaID)

IApeClass *pgc = 0;
// find the class object via CoGetClassObject
// находим объект класса с помощью CoGetClassObject
HRESULT hr = CoGetClassObject(CLSIDGorilla , CLSCTXALL , 0, IIDIApeClass ,
(void**)&pgc);
if (SUCCEEDED(hr))

IApe *pApe = 0;
// use the class object to find an existing gorilla
// используем объект класса для нахождения существующей гориллы
hr = pgc->GetApe(nGorillaID, &pApe);
if (SUCCEEDED(hr))

// tell the designated gorilla to eat a banana
// прикажем указанной горилле есть бананы
hr = pApe->EatBanana();
pApe->Release();

pgc->Release();

return hr;
```

Данный пример использует объект класса для того, чтобы *Gorilla* нашла именованный объект и проинструктировала его есть бананы. Чтобы этот пример работал, нужно, чтобы какой-нибудь внешний посредник дал вызывающему объекту имя какой-нибудь известной гориллы. Можно построить пример и таким образом, чтобы любая неизвестная горилла могла быть использована для удовлетворения запроса:

```
HRESULT CreateAGorillaAndEatBanana(void)

IApeClass *pgc = 0;
// find the class object via CoGetClassObject
// находим объект класса с помощью CoGetClassObject
HRESULT hr = CoGetClassObject(CLSIDGorilla , CLSCTXALL , 0, IIDIApeClass ,
(void**)&pgc);
if (SUCCEEDED(hr))

IApe *pApe = 0;
// use the class object to create a new gorilla
// используем объект класса для создания новой гориллы
hr = pgc->CreateApe(&pApe);
if (SUCCEEDED(hr))

// tell the new gorilla to eat a banana
// прикажем новой горилле есть бананы
hr = pApe->EatBanana();
pApe->Release();

pgc->Release();
```

```
return hr;
```

Отметим, что за исключением специфического использования метода *IApeClass* , эти примеры идентичны. Поскольку объекты класса могут экспортировать сколь угодно сложные интерфейсы, то их можно использовать для моделирования довольно изощренной стратегии активации, инициализации и размещения объектов.

Классы и серверы

COM-сервер – это двоичный файл, содержащий код метода для одного или более COM-классов. Сервер может быть упакован или в динамически подключаемую библиотеку (DLL), или в нормальный исполняемый файл. В любом случае за загрузку любого типа сервера автоматически отвечает диспетчер управления сервисами SCM.

Если в запросе на активацию объекта указана внутрипроцессная активация, то вариант сервера на базе DLL должен быть доступен для загрузки в адресном пространстве клиента. Если же в запросе на активацию указаны внепроцессная или внехостовая активация, то для запуска серверного процесса на указанной хост-машине (она может совпадать с машиной клиента) будет использован исполняемый файл. COM поддерживает также выполнение DLL-серверов в суррогатных процессах (*surrogate processes*) с целью разрешить использование внепроцессной и внехостовой активации существующих внутрипроцессных серверов. Подробности того, как суррогатные процессы связаны с внепроцессной и внехостовой активацией, будут изложены в главе 6.

Чтобы клиенты могли активировать объекты, не беспокоясь о том, как упакован сервер или где он инсталлирован, в COM предусмотрена конфигурационная база данных, отображающая *CLSID* на тот сервер, который реализует этот класс. При использовании версий Windows NT 5.0 или выше основным местом расположения этой конфигурационной базы данных является директория NT (NT Directory). Эта директория является рассредоточенной защищенной базой данных, в которой хранится служебная информация об учетных записях пользователей, хост-машинах и прочее. С тем же успехом в директории NT можно хранить информацию и о COM-классах. Эта информация записывается в области директории, называемой *COM Class Store* (хранилище COM-классов). COM использует Class Store для перевода *CLSID* в файлы реализации (в случае локальных запросов на активацию) или в удаленные хост-имена (в случае удаленных запросов на активацию). Если запрос на активацию для *CLSID* сделан на данной машине, то в первую очередь опрашивается локальный кэш. Если в локальном кэше нет доступной конфигурационной информации, то COM посылает запрос в Class Store о том, чтобы реализация стала доступной из локальной машины. Это может просто означать добавление некоторой информации в локальный кэш, чтобы переадресовать запрос на другую хост-машину, или же это может привести к загрузке реализации класса на локальную машину и к запуску программы инсталляции. В любом случае, если класс зарегистрирован в Class Store, он доступен для запроса на активацию со стороны клиента в рамках ограничений безопасности.

Локальный кэш, упоминавшийся при обсуждении Class Store, официально называется системным реестром, или базой конфигурации системы (Registry). Реестр является иерархической базой данных, хранящейся в файлах на каждой машине, которую COM использует для преобразования *CLSID* в имена файлов (в случае локальной активации) или удаленные имена хостов (в случае удаленной активации). До Windows NT 5.0 реестр был единственным местом размещения конфигурационной информации COM. Быстрый поиск в реестре может быть осуществлен с помощью иерархических *ключей (keys),* имена которых представляют собой строки, разделенные обратными косыми чертами. Каждый ключ в

реестре может иметь одно или несколько *значений,* которые могут иметь в своем составе строки, целые значения или двоичные данные. В реализации COM на Windows NT 4.0 большая часть ее конфигурационной информации записывается под именем

HKE*YLOCAL* MACHINE

в то время как большинство программ используют более удобный псевдоним

HKEYCLASSESROOT

Реализация COM на Windows NT 5.0 продолжает использовать HKEYCLASSESROOT для установок в рамках всей машины, но также разрешает каждому пользователю назначить свою конфигурацию CLSID для обеспечения большей безопасности и гибкости. Под Windows NT 5.0 COM вначале опрашивает

HKEYCURRENTUSER

прежде чем опрашивать HKEYCLASSESROOT. Для удобства записи часто используются аббревиатуры HKLM, HKCR и HKCU вместо HKEYLOCALMACHINE, HKEYCLASSESROOT и HKEYCURRENTUSER , соответственно[26].

COM хранит информацию, относящуюся к *CLSID* всех машин, под ключом реестра HKCR

В версии Windows NT 5.0 или выше COM ищет информацию о классах каждого пользователя под ключом HKCU

Под одним из этих ключей будет сохранен список локальных *CLSID* , для каждого *CLSID* – свой подключ. Например, класс *Gorilla* , использовавшийся ранее в этой главе, мог бы иметь по всей машине запись по подключу[27]:

```
[HKCR{571F1680-CC83-11d0-8C48-0080C73925BA]
@="Gorilla"
```

Для обеспечения локальной активации объектов *Gorilla* запись для *CLSID Gorilla* в реестре должна иметь подключ, показывающий, какой файл содержит исполняемый код для методов класса. Если сервер упакован как DLL, то требуется такая запись:

```
[HKCR{571F1680-CC83-11d0-8C48-0080C73925BA
```

---

[26] *1* Эти аббревиатуры не допускаются в исходном коде или в конфигурационных файлах. Они просто дают возможность длинным именам ключей фигурировать в виде одной строки без разделителей в документации или других текстах о COM. Читателю следует раскрывать аббревиатуры при чтении вслух или при использовании в исходном коде.

[27] *2* Приведенный здесь способ записи использует стандартный синтаксис *REGEDIT4* . Строки, содержащиеся внутри скобок, соответствуют именам ключей. Пары *имя=значение (name = value)* под ключом обозначают значения, присвоенные указанному ключу. Необычное имя "@" показывает значение ключа по умолчанию.