ВВедение В COM

АВтор: *Том Армстронг*
Источник: *"ActiveX-Создание Web-приложений", глава 6*
Материал предоставил: *ЕВгений Щербатов*

Часть 1. Общие сведения и немного истории.

Технологии ActiveX охВатыВают обширные области программного обеспечения. Корпорация Microsoft применяет их поВсеместно В сВоих продуктах. Они также интенсивно используются В операционных системах. В этой главе мы рассмотрим технологию, которая лежит В основе стандартоВ OLE и ActiveX - модель компонентного объекта (Component Object Model - COM) Для лучшего понимания сущности этой модели будет разработан простой COM-компонент. Сначала мы напрямую Воспользуемся Возможностями языка C++, а затем перейдем к использоВанию библиотеки активных шаблоноВ (Active Template Library - ATL), которая представляет собой множестВо шаблоноВ языка C++, предназначенных для построения эффективных COM-компонентоВ.

Изложенное В этой главе послужит нескольким целям. Во-первых, будет ВВедено понятие COM. Во-Вторых, Вы получите представление о сущности этой технологии и Возможностях ее применения В собственных приложениях. И, В-третьих, данный материал послужит ВВедением В технологии, В основе которых лежит COM. Они рассматриваются В последующих главах.

Модель компонентного объекта

Модель компонентного объекта фирмы Microsoft яВляется, как следует из её названия, *моделью* для проектирования и создания компонентных объектоВ. Модель определяет множестВо технических приемоВ, которые могут быть использоВаны разработчиком при создании незаВисимых от языка программных модулей, В которых соблюдается определенный дВоичный стандарт. Корпорация Microsoft обеспечиВает реализацию модели COM Во Всех сВоих Windows-средах. В других операционных средах, таких как Macintosh и UNIX, технология COM также поддерживается, но не обязательно средствами фирмы Microsoft.

Технологии OLE и ActiveX яВляются Высокоуровневыми программными сервисами, построенными фирмой Microsoft на основе технологии COM. Доступ к средствам COM, OLE и ActiveX обеспечиВается операционной системой Windows посредством набора COM-интерфейсоВ и небольшого числа функций WIN32 API. Как и В случае с COM, В отличных от Windows операционных системах также поддерживаются сервисы подобные OLE и ActiveX.

> ПРИМЕЧАНИЕ
> Большинство технологий ActiveX фирмы Microsoft переносимы В другие, отличные от Windows операционные среды. Фирма Microsoft передала спецификации и руководства по реализации технологий ActiveX другим группам разработчикоВ. Новая группа, называемая Active Group, Вместе с так называемой "группой открытых технологий" (Open Group) В настоящее Время работает над превращением технологии ActiveX В открытый стандарт. За дополнительной информацией обращайтесь по адресу: http://www.activex.org.

ДВоичный стандарт (или незаВисимость от языка программирования)

Одной из наиболее Важных черт COM яВляется ее способность предоставлять дВоичный стандарт для программных компонентоВ. Этот

двоичный стандарт обеспечивает средства, с помощью которых объекты и компоненты, разработанные на разных языках программирования разными поставщиками и работающие в различных операционных системах, могут взаимодействовать без каких-либо изменений в двоичном (исполняемом) коде. Это является основным достижением создателей COM и отвечает насущным потребностям сообщества разработчиков программ.

Многоразовое использование программного обеспечения является одной из первоочередных задач при его разработке и обеспечивается составляющими его модулями, которые должны работать в разнообразных средах. Обычно программное обеспечение разрабатывается с использованием определенного языка программирования, например C++, и может эффективно применяться только в том случае, если другие разработчики компонентов также применяют C++.

Например, если мы разрабатываем C++ - класс, предназначенный для манипулирования с данными, то необходимым условием его использования в других приложениях является их разработка на языке C++. Только C++ - компиляторы могут распознать C++ - классы. Фактически, поскольку средства C++ не поддерживают никакого стандартного способа адаптации вызовов C++ - функций к новой программной среде, использование программного обеспечения в этой новой среде требует применения такого же (или аналогичного) инструментального средства для его обработки. Другими словами, использование класса в другой операционной среде требует обязательного переноса в эту среду исходного текста программы данного класса.

Применение двоичного кода позволяет разработчику создавать программные компоненты, которые могут применяться без использования языков, средств и систем программирования, а только с помощью *двоичных компонентов* (например, DLL- или EXE - файлов). Эта возможность является для разработчиков очень привлекательной. Ведь теперь они могут выбирать наиболее удобный для себя язык и средство разработки компонентов, не заботясь о языке и средствах, которые будет использовать другой разработчик.

Чтобы получить представление о том, как соотносятся между собой C++ - объекты и COM - компоненты, рассмотрите табл. 1. В данный момент эти соотношения, быть может, и не имеют для Вас глубокого смысла, но они дают понять, что C++ и COM представляют собой две совершенно разные вещи. C++ - это язык, а COM - технология.

C++-объект (экземпляр класса)	COM-объект
Позволяет использовать только один общий интерфейс, представляющий собой множество C++-методов.	Обычно предоставляет более одного общего интерфейса
Зависит от языка программирования.	Обеспечивается независимость от языка - COM-объекты реализуются и используются в различных языках программирования.
Отсутствует встроенная возможность проверки версии.	Поддерживается встроенный способ проверки версий объектов. Обеспечивается независимость от местоположения на жестком диске.

Таблица 1. Сравнение объектов C++ и COM

Независимость от местоположения

Другое важное свойство COM известно под названием *независимости от местоположения* (Location Transparency). Независимость от местоположения означает, что пользователь компонента, клиент, не обязательно должен знать, где находится определенный компонент. Клиентское приложение использует одинаковые сервисы COM для создания экземпляра и использования компонента независимо от его фактического расположения. Компонент может находиться непосредственно в адресном пространстве задачи клиента (DLL-файл), в пространстве другой задачи на том же компьютере (EXE-файл) или на компьютере, расположенном за сотни миль (распределенный объект). Технологии COM и DCOM (Distributed COM - распределенная COM) обеспечивают независимость от местоположения. Другими средствами, реализующими эту способность, являются *сервисы распределенных объектов*. Аналогичные возможности обеспечивает стандарт CORBA. Поскольку клиентское приложение взаимодействует с COM - компонентами, вне зависимости от их положения, одинаковым образом, интерфейс клиента тоже не меняется. Независимость от местоположения позволяет разработчику создавать масштабируемые приложения.

Понятия OLE и ActiveX

Термин OLE имеет длинную историю. Вначале (приблизительно с 1991 года) он представлял собой сокращение слов *Object Linking and Embedding* (связывание и внедрение объектов). Первоочередной задачей OLE в то время была поддержка связывания и внедрения объектов в Windows-приложениях. Возможности получаемых таким образом *составных документов* позволяли пользователям внедрять электронные таблицы Excel непосредственно в документы Word и т.д.

Впоследствии, с выходом версии OLE 2.0 (приблизительно в 1993 году), термин OLE перестал употребляться как просто сокращение и вобрал в себя понятия нескольких технологий, основанных на модели компонентного объекта. Многие новые возможности OLE не имеют ничего общего со связыванием и внедрением. Хорошим примером этого является OLE-автоматизация (или просто автоматизация), вообще не имеющая отношения к составным документам. Основной задачей OLE-автоматизации является обеспечение взаимодействия компонентов и приложений независимо от языков программирования и средств разработки.

В апреле 1996 года Microsoft приняла в сферу своих интересов среду WWW и ввела в обращение термин *ActiveX*, призванный отобразить новое направление в стратегии выпуска программных продуктов фирмы. Однако большинство новых технологий ActiveX уже существовали до апреля 1996 года под другим названием: OLE. В общем, термин *ActiveX* заменил термин *OLE*, цель этой замены - подчеркнуть превосходство COM-технологий фирмы Microsoft. Теперь термин OLE снова может быть использован для описания тех технологий, которые относятся к составным документам, а также связыванию и внедрению объектов.

Средства, основанные на COM

Все средства разработки Microsoft в той или иной степени поддерживают технологии COM, OLE и ActiveX. В этой главе мы будем использовать язык C++, а немного позже - библиотеку активных шаблонов Microsoft (ATL). Однако все высокоуровневые средства и языки, такие как Visual Basic, C++ и Java, обеспечивают хорошую поддержку компонентов COM.

Часть 2. COM-объекты, интерфейсы, C++ и о том, как это работает.

Язык C++, компоненты и интерфейсы

Одним из главных преимуществ разработки с помощью объектно-ориентированных языков, таких как C++ и Java, является возможность эффективной инкапсуляции внутренних функций и данных. Это осуществимо именно благодаря объектной ориентированности этих языков. В объекте скрыты способы его реализации, а "наружу" предоставляется только хорошо определенный интерфейс, позволяющий внешним клиентам эффективно использовать функциональные возможности объекта. Технология COM обеспечивает эти возможности также с помощью определения стандартных способов реализации и предоставления интерфейсов COM-объекта.

Далее мы приведем определение класса для простого компонента, который будет построен в этой главе. Начнем с обыкновенного C++-класса, а затем преобразуем его в COM-объект. Особой необходимости строить COM-объекты с помощью C++ нет, но, как будет видно из дальнейшего, некоторые технические приемы C++ находят применение и при создании COM-компонентов.

```
class Math
{
// описание интерфейса
public:
   long Add( long Op1, long Op2 );
   long Subtract( long Qp1, long Op2 );
   long Multiply( long Op1, long Op2 );
   long Divide( long Op1, long Op2 );

private:
   // реализация
   string m_strVersion;
   string get_Version( );
};

long Math::Add( long Op1, long Op2 )
{
   return Op1 + Op2;
}

long Math::Subtract( long Op1, long Op2 )
{
   return Op1 - Op2;
}
long Math::Multiply( long Op1, long Op2 ) {
   return Op1 * Op2;
}
long Math::Divide( long Op1, long Op2 )
{
   return Op1 / Op2;
}
```

Этот класс поддерживает выполнение основных математических операций. Вы передаете объекту два числа, а он будет прибавлять, вычитать либо делить их и возвращать результат. Нашей задачей в данной главе будет превращение указанного класса в COM-объект, который можно будет использовать в любом языке программирования, поддерживающем COM-интерфейс. Первым шагом будет определение для компонента интерфейса с помощью абстрактного класса такого вида:

```
class IMath
{
public:
   virtual long Add(long Op1, long Op2) = 0;
   virtual long Subtract(long Op1, long Op2) = 0;
   virtual long Multiply(long Op1, long Op2) = 0;
   virtual long Divide (long Op1, long Op2) = 0;
};
```

Затем мы создадим производный класс и опишем его методы точно так же, как это делалось раньше:

```
class Math : public IMath
{
public:
   long Add(long Op1, long Op2);
   long Subtract(long Op1, long Op2);
   long Multiply(long Op1, long Op2);
   long Divide (long Op1, long Op2);
};
```

Это лишь первый шаг в преобразовании класса C++ в объект, доступный для программ, написанных на других языках. COM-технологии, такие как OLE и ActiveX, в основном реализуются посредством интерфейсов типа рассмотренного нами ранее класса IMath. Наш новый класс является абстрактным, т.е. он содержит, по крайней мере, одну чисто виртуальную функцию и одни лишь методы (без каких-либо данных) компонентного объекта.

Начальная буква "I" в названии класса IMath отражает тот факт, что он является объявлением интерфейса. Такие обозначения используются в технологии программирования COM повсеместно. Класс IMath объявляет внешний интерфейс для компонента Math. Наиболее важным аспектом этого нового класса является его способность порождать виртуальную таблицу функций C++ (Vtable) в любом производном классе.

Использование виртуальных функций в базовом классе является центральным моментом в проектировании COM-компонентов. Определение абстрактного класса порождает таблицу, содержащую только открытые методы (т.е. интерфейс) класса. Класс IMath не содержит переменных-членов и функций реализации объекта. Его единственной задачей является порождение производного класса Math для виртуальной реализации методов интерфейса компонента.

В технологии COM доступ к компонентам обеспечивается только с помощью указателей на виртуальные таблицы. Таким образом, прямой доступ к конкретным данным компонента становится невозможным. Внимательно изучите наш пример. Он достаточно прост, но отражает ключевую концепцию COM – использование виртуальных таблиц для доступа к функциональным возможностям компонента. Наконец, COM-интерфейс представляет собой просто указатель на указатель *виртуальной таблицы* (virtual table, или Vtable) C++. На рис. 1 отражено это взаимоотношение для случая компонента Math.

В нашем примере имеется несколько моментов, которые должны быть хорошо поняты. Во-первых, все COM-технологии, такие как ActiveX и OLE, содержат множество определений абстрактного интерфейса типа нашего класса IMath. В конечном счете, деятельность разработчика состоит в конкретной *реализации* этих интерфейсов. Это одна из причин того, почему ActiveX является стандартом. Технология ActiveX обеспечивает объявление интерфейса, а Вы, разработчик, обеспечиваете его реализацию. Таким образом, несколько разработчиков могут по-разному реализовать стандартный компонент ActiveX. Эта концепция лежит в основе элементов управления и всех технологий ActiveX. Спецификация ActiveX определяет абстрактные классы, которые требуется реализовать для получения полноценного элемента управления ActiveX.

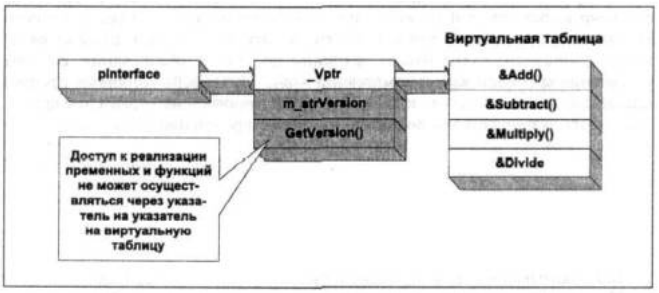

Рис. 1. Виртуальная таблица функций и COM-интерфейс

Во-Вторых, Microsoft дВижется В направлении от создания библиотек функций (например, Win32 API) к созданию компонентоВ операционной системы, предоставляющих определенные COM-интерфейсы. В таком случае Вы становитесь пользоВателем компонента. Вместо функций API Вы получаете интерфейс для компонента системы Windows и доступ к его функциям посредстВом COM-интерфейса. Это дает Возможность заменять реализацию компонента операционной системы, не оказыВая ВоздействиЯ на другие компоненты и приложения.

В-третьих, Во многих случаях Вы будете одноВременно Выступать В качестВе производителя и потребителя COM-интерфейсоВ. Технологии ActiveX используют их очень широко. ВзаимодействиЕ между компонентами ActiveX достигается за счет интерфейсоВ. Запуск и согласоВание приложений осуществЛяется через интерфейсы. Фактически, COM-интерфейсы присутстВуют ВезДе.

Стандартные COM-интерфейсы

Теперь Вы поняли, что технология COM предусматриВает наличие множестВа абстрактных классоВ, которые требуют реализации. При построении COM-компонента первым делом нужно реализоВать интерфейс, который должны использоВать Все COM-компоненты: **IUnknown**. Компонент должен не только реализоВать интерфейс **IUnknown** для себя самого, но и обеспечить его реализацию для каждого сВоего интерфейса. Вначале это может показаться Вам слишком сложным, но именно так и обстоят дела. БольшинстВо COM-компонентоВ предлагают несколько интерфейсоВ, и запомните: COM- интерфейс – это просто указатель на C++-интерфейс. Более подробно мы обсудим это чуть позже.

Интерфейс **IUnknown** Выполняет дВе функции. ПерВая состоит В том, чтобы обеспечить стандартный способ запроса определенного интерфейса данного компонента его пользоВателем (клиентом). Эту Возможность предоставляет метод **QueryInterface**. ВтораЯ функция состоит В обеспечении способа управления ВременеМ жизни компонента изВне. Интерфейс **IUnknown** предоставляет дВа метода (**AddRef** и **Release**), обеспечиВающих управление ВременеМ жизни экземпляра компонента. ПриВедем определение **IUnknown**.

```
class IUnknown
{
public:
   virtual HRESULT QueryInterface(REFID riid, void** ppv)=0;
   virtual ULONG AddRef () = 0;
   virtual ULONG Release() = 0;
};
```

Как указыВалось Выше, поскольку **IUnknown**, яВляется объЯВлением COM-интерфейса, он представляет собой абстрактный класс, любой производный класс которого должен реализоВать три ранее описанных метода и, таким образом, занести их В Виртуальную таблицу. Прежде чем пойти дальше, остановимся на рассмотрении типа ВозВращаемого функцией **QueryInterface** значения, которым яВляется тип **HRESULT**.

Тип данных HRESULT

БольшинстВо методоВ COM-интерфейса и функции API ВозВращают значение типа HRESULT (исключениями яВляются AddRef и Release). В Win32 тип данных HRESULT определяется как DWORD (32-битоВое целое), а ВозВращаемое значение этого типа содержит информацию о результате ВызоВа функции. Старший бит сигнализирует об успешном или ошибочном заВершении работы функции, а следующие 15 битоВ идентифицируют тип ошибки и обеспечиВают способ группироВки однотипных кодоВ заВершения; младшие 16 битоВ предоставляют специфическую информацию о происшедшем. Структура данных HRESULT идентична структуре значений флагоВ статуса, используемых функциями Win32 API.

Система разработки COM обеспечиВает несколько макросоВ, помогающих узнать результат ВызоВа метода. Макрос SUCCEEDED принимает значение TRUE при успешном ВызоВе функции, а макрос FAILED принимает это же значение при ошибке ВызоВа функции. НазВанные макросы не яВляются специфичными для COM и ActiveX, а используются повсеместно В среде Win32 и определяются В файле WINERROR.H. ВозВращаемые значения В Win32 предВаряются префиксом S_ В случае нормального заВершения и префиксом E_ – В случае ошибки.

Метод IUnknown::QueryInterface

Метод QueryInterface обращается к идентификатору интерфейса (Interface Identifier – IID), который представляет собой 128-битоВый уникальный идентификатор (а именно GUID, о котором Вскоре пойдет речь), и ВозВращает указатель на определенный интерфейс (например, IUnknown, IMath), предоставляемый COM-объектом. Указатель ВозВращается через ВторОй параметр, который яВляется указателем на указатель типа void.

ВозВратиВшись, к примеру, IMath, Вы можете легко убедиться В том, что компонент Math реализует IUnknown В каждом сВоем интерфейсе. В настоящий момент у нас имеется только один интерфейс IMath, поэтому следует сделать следующее:

```
class IMath : public IUnknown
{
public:
   virtual long Add(long Op1, long Op2) = 0;
   virtual long Subtract(long Op1, long Op2) = 0;
   virtual long Multiply(long Op1, long Op2) = 0;
   virtual long Divide (long Op1, long Op2) = 0;
};
class Math : public IMath
{
public:
```

```
    HRESULT QueryInterface(REFIID riid, void** ppv);
    ULONG   AddRef();
    ULONG   Release();

    long Add(long Op1, long Op2);
    long Subtract(long Op1, long Op2);
    long Multiply(long Op1, long Op2);
    long Divide (long Op1, long Op2);
};
```

Теперь мы получили полное объявление для COM-компонента. Если пользователю потребуется вызвать функции компонента Math, он запросит интерфейс IMath; однако, поскольку все COM-компоненты обеспечивают интерфейс IUnknown, пользователь в первую очередь должен будет запросить указатель на IUnknown и лишь затем – с помощью метода IUnknown::QueryInterface() – интерфейс IMath. Теперь нам нужно реализовать семь методов, унаследованных от классов IUnknown и IMath. В результате виртуальная таблица примет вид, представленный на рис. 2.

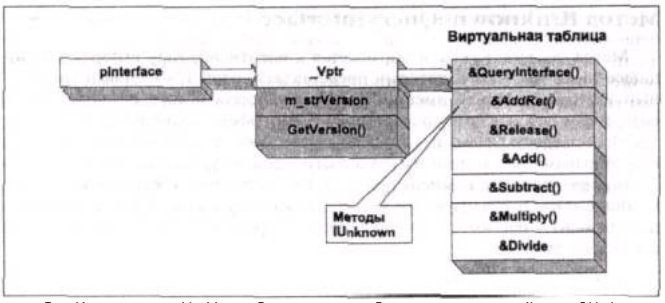

Рис. 2. Компонент Math с включенным в него интерфейсом IUnknown

Методы IUnknown::AddRef() и IUnknown::Release()

За управление временем жизни компонентов отвечают два метода интерфейса IUnknown: AddRef() и Release(). Обычно COM-компонент имеет несколько интерфейсов, каждый из которых может быть связан со многими внешними клиентами. Обратите внимание на то, что в нашем примере компонент в действительности является классом C++, а в данный момент мы обсуждаем управление временем жизни определенного экземпляра класса. Пользователь будет создавать экземпляр с помощью некоторого механизма, который мы еще обсудим, и использовать возможности этого экземпляра посредством его COM-интерфейсов. Первоначально экземпляр будет создан с помощью оператора C++ **new**, а затем мы попытаемся определить, когда этот экземпляр может быть удален.

Поскольку экземпляр COM-компонента может иметь несколько интерфейсов, связанных со многими клиентами, нашему объекту необходимо иметь некоторую возможность подсчета обращений к нему (счетчик). Всякий раз, когда клиент запрашивает интерфейс, значение счетчика будет увеличиваться, а когда клиент завершает работу с интерфейсом – уменьшаться. В конце концов, когда значение счетчика обращений станет равным нулю, COM-компонент будет уничтожен. Именно для этого и служат методы IUnknown::AddRef() и IUnknown::Release().

Таким образом, в рассматриваемом нами примере класса Math требуется отслеживать значение внутреннего счетчика обращений. Назовем этот счетчик m_lRef. Когда компонент по запросу клиента возвращает интерфейс, значение счетчика будет увеличиваться, а по окончании использования интерфейса клиент уменьшает это значение посредством вызова IUnknown::Release().

Пользователь компонента не может непосредственно удалить экземпляр C++-объекта, поскольку в его распоряжении имеется только указатель на виртуальную таблицу функций C++. В действительности клиент не имеет права удалять объект в любом случае, поскольку могут существовать другие клиенты, использующие тот же компонентный объект. Только сам компонент, основываясь на значении своего внутреннего счетчика обращений, может определить момент своего удаления. Ниже приводится текст программы компонента **Math** с включением реализации интерфейса **IUnknown**.

```
class Math : public IMath
{
public:
    HRESULT QueryInterface(REFIID riid, void** ppv);
    ULONG   AddRef();
    ULONG   Release();

    long Add(long Op1, long Op2);
    long Subtract(long Op1, long Op2);
    long Multiply(long Op1, long Op2);
    long Divide (long Op1, long Op2);

    // Реализация
private:
    // Новая переменная-член класса добавлена для подсчета
    // обращений извне к интерфейсу объекта
    DWORD m_lRef;

public:
    Math( );
};

Math::Math( )
{
    m_lRef = 0;
}
HRESULT Math::QueryInterface( REFIID riid, void** ppv )
{
    switch( riid )
    {
    case IID_IUnknown:
    case IID_IMath;
        *ppv = this;
```

```
      // Поскольку мы возвращаем новый указатель на
      // интерфейс, необходимо вызвать метод AddRef
      AddRef();
      return ( S_OK ) ;

   default:
      return ( E_NOINTERFACE );
   }
}

// Реализация функции IUnknown:Release
ULONG Math::Release( )
{
   InterlockedDecrement( &m_lRef );

   // когда значение счетчика обращений
   // становится равным нулю, объект удаляет сам себя
   if ( m_lRef == 0 )
   {
      delete this;
      // нельзя вернуть m_lRef, поскольку его уже не существует
      return 0;
   }
   else
      return m_lRef;
}

// Реализация функции IUnknown::AddRef
ULONG Math::AddRef( )
{
   InterlockedIncrement( &m_lRef );
   return m_lRef;
}
```

Для реализации функций AddRef() и Release(), унаследованных от класса IUnknown, мы ввели переменную-член m_lRef, которая отвечает за подсчет текущих обращений к объекту или ожидающих обработки указателей интерфейса. Хотя функции AddRef () и Release () могут изменять значение счетчика обращений к COM-интерфейсу, сам интерфейс не является экземпляром объекта. Объект в каждый момент времени может иметь любое количество пользователей своих интерфейсов и должен отслеживать значение внутреннего счетчика активных интерфейсов. Если это значение достигает нуля, объект сам себя удаляет.

Очень важно правильно и своевременно использовать методы AddRef () и Release(). Эта пара функций аналогична паре операторов new и delete, используемых в языке C++ для управления памятью. Как только пользователь получает новый указатель на интерфейс или присваивает его значение какой-либо переменной, необходимо вызывать AddRef (). В данном случае следует быть очень внимательным, поскольку некоторые функции COM-интерфейсов возвращают указатели на другие интерфейсы и в таких случаях сами вызывают метод AddRef() для возвращаемого указателя. Наиболее наглядным примером этого является метод QueryInterface(), в котором AddRef () вызывается при каждом запросе интерфейса и, таким образом, не возникает необходимость в новом вызове метода AddRef ().

> **ПРИМЕЧАНИЕ**
> В предыдущем примере были использованы библиотечные функции Win32 API InterlockedIncrement и InterlockedDecrement. Это обеспечивает возможность синхронизированного доступа к внутренним счетчикам и делает компонент безопасным в отношении потоков (по крайней мере, с точки зрения управления временем жизни компонента).

Множественные интерфейсы

Одним из наиболее мощных свойств COM является то, что каждый компонент может предоставлять и обычно предоставляет несколько интерфейсов для одного объекта. Подумайте над этим еще раз. При разработке C++-класса создается всего один интерфейс. А при создании на основе класса компонента создается как минимум еще один интерфейс. Обычно при построении COM-компонента Вам необходимо предоставить несколько интерфейсов для одного экземпляра класса C++.

Объявление нескольких интерфейсов для одного класса C++ на первый взгляд не является трудным, но на самом деле может быть очень непростым делом. Во-первых, поскольку COM-интерфейсы фактически являются указателями на виртуальную таблицу функций C++, класс с множеством интерфейсов требует создания множества виртуальных таблиц. Другой причиной взаимосвязи компонентных объектов и их классов реализации интерфейса является необходимость подсчета количества обращений. Все интерфейсы COM-объекта должны взаимодействовать между собой в обеспечение подсчета обращений. Для каждого объекта существует только один счетчик обращений, и интерфейсы должны использовать его совместно. Эта взаимосвязь осуществляется с помощью множества интерфейсов IUnknown. Запомните, каждый COM-компонент должен обладать собственной реализацией интерфейса IUnknown.

В языке C++ существуют три основных способа обеспечения множества интерфейсов для COM-компонента: множественное наследование (Multiple Inheritance), реализации интерфейсов (Interface Implementations), вложенность классов C++ (C++ class nesting). Мы сосредоточимся на одном из этих трех способов – множественном наследовании классов C++, так как именно с помощью него в активной библиотеке шаблонов (ATL) реализованы COM-интерфейсы. Вложенность классов C++ применяется и в библиотеках базовых классов Microsoft (MFC), речь о которых пойдет в последующих главах.

Язык C++ и множественное наследование

В примере компонента Math мы использовали один интерфейс IMath. Однако, большинство COM-компонентов предоставляют несколько интерфейсов. Например, предположим, что компонент Math предоставляет еще и интерфейс анализатора выражений, Вот примерное определение этого интерфейса:

```
class IExpression : public IUnknown
{
public:
   virtual void SetExpression(string strExpression) = 0;
   virtual BOOL Validate() = 0;
   virtual BOOL Evaluate() = 0;
```

```
};
```

Используя множественное наследование, мы могли бы снабдить класс Math обоими интерфейсами:

```
class Math : public IMath, public IExpression
{
// Реализация компонента включает
// и реализацию каждого наследуемого интерфейса
...
};
```

Единственная проблема этого подхода состоит в том, что в таком случае может возникнуть конфликт базовых интерфейсов IUnknown (так как оба класса, и IMath, и IExpression, наследуются от IUnknown), однако в данном случае ничего подобного не происходит, поскольку унаследованный класс должен "совместно использовать" реализацию интерфейса IUnknown. Важным моментом здесь является то, что COM-компоненты должны предоставлять несколько интерфейсов или, другими словами, несколько указателей на виртуальные таблицы. Если обеспечивать это с помощью множественного наследования, то необходимо, чтобы тип указателя данного экземпляра правильно приводился к типу указателя на виртуальную таблицу. Таким образом, метод QueryInterface для описанного выше класса будет иметь вид:

```
HRESULT Math::QueryInterface(REFIID riid, void** ppv)
{
   switch(riid)
   {
      case IID_IUnknown:
      case IID_IMath:
         // Множественное наследование требует явного приведения типов
         *ppv = (IMath*) this;
         break;
      case IID_IExpression:
         // Множественное наследование требует явного приведения типов
         *ppv = (IExpression*) this;
         break;
      default:
         return(E_NOINTERFACE) ;
   }

   // Возвращаем указатель нового интерфейса
   // и таким образом вызываем AddRef для нового указателя
   AddRef();
   return(S_OK);
}
```

Указатель на интерфейс IUnknown может быть возвращен посредством обращения к IMath или IExpression, поскольку оба эти класса содержат такой метод. Мы выберем обращение к IMath. Поддержка нескольких интерфейсов объекта представляет собой одну из наиболее привлекательных особенностей COM. Мы еще вернемся к этой теме при обсуждении активной библиотеки шаблонов.

Часть 3. COM-серверы, какие они бывают и о том, что дал нам Microsoft, чтобы управлять ими.

Хранение компонентов

До сих пор мы обсуждали требования, предъявляемые к интерфейсам COM-компонентов. Однако после реализации компонента на том или ином языке программирования (в нашем случае на C++) он должен выполняться как процесс в определенной операционной системе. COM-компоненты хранятся либо в исполняемых файлах (EXE), либо в файлах динамически загружаемых библиотек Windows (DLL), либо в тех и в других. Каждый вариант хранения имеет свои преимущества, и разработчик компонентов должен реализовывать определенные функции различным образом в зависимости от выбранного им способа размещения компонента.

Термин *локальный сервер* используется для описания компонентов, которые хранятся в исполняемых файлах. Такие файлы, помимо COM-компонентов, могут содержать и другие функции. Например, Microsoft Word является локальным сервером. Он не только обеспечивает возможности обработки текстов, но также предоставляет множество COM-компонентов, к которым имеют доступ другие приложения.

Внутризадачный сервер представляет собой DLL-файл Windows. Выполнение этого компонента происходит в контексте вызывающего процесса и, таким образом, процесс клиента имеет прямой доступ к любому DLL-компоненту. Понятие внутризадачности сервера приобретет особую значимость при рассмотрении пользовательских COM-интерфейсов и процессов транспортировки.

Удаленный сервер представляет собой компонент, загружаемый и выполняемый на удаленном компьютере. Обычно удаленный сервер строится как исполняемый файл, однако это не является обязательным требованием. Доступ к компонентам, хранящимся только в DLL-файлах, может осуществляться также и в удаленном режиме. В таких случаях модель COM создает *суррогатный процесс*, в котором могут выполняться удаленные DLL-файлы.

> **ПРИМЕЧАНИЕ**
> Предоставляемая моделью DCOM поддержка внутризадачных суррогатных процессов обеспечивается только при наличии среды NT 4.0, расширенной за счет использования пакета Service Pack 2.

Одним из преимуществ COM является независимость клиента (пользователя компонента) от местоположения компонента. Как указывалось выше, COM-сервисы могут быть реализованы в трех различных конфигурациях: внутризадачной посредством DLL-файлов на локальном компьютере, межзадачной на одном и том же компьютере (локальный сервис) или на удаленном компьютере посредством DLL-файла или исполняемого кода. Однако клиенту не требуется знать, каким образом реализован компонент или где находятся необходимые сервисы. Он создает экземпляр компонента, а расположение и запуск возлагаются на средства COM. Таким образом, многосвязность является неотъемлемым свойством COM-компонентов.

При определении способа реализации компонентов следует принять во внимание два основных фактора. Первый из них – эффективность. Поскольку внутризадачные серверы выполняются в адресном пространстве клиента, они являются наиболее

эффективными. Кроме того, в данном случае не требуется никаких дополнительных средств для передачи параметров методам компонента.

Транспортировка

Транспортировка (marshalling) – это процесс передачи аргументов функций и возвращаемых значений через "границу" процесса или по сети. Для этого требуется скопировать передаваемые значения в общую память таким образом, чтобы другой процесс получил к ним доступ. Транспортировка внутренних типов данных, таких как short и long, не представляет сложностей, но для большинства значений других типов, таких как указатели и структуры, транспортировка осуществляется немного сложнее. Нельзя создать копию указателя, поскольку его значение (адрес) не имеет смысла в контексте другого процесса. В этом случае приходится копировать всю структуру данных, адрес которой хранится в указателе. Только тогда другой процесс сможет получить доступ к этим данным. Поскольку внутризадачные серверы выполняются в адресном пространстве процесса клиента, для них не требуется никакого специального механизма транспортировки (хотя границы потока преодолеваются). Поэтому внутризадачные серверы обеспечивают наиболее эффективный способ взаимодействия с COM-компонентами. Наш первый пример компонента будет реализован как внутризадачный сервер. Таким образом, нам не придется беспокоиться о предоставлении поддержки для транспортировки данных.

Если компонент является внутризадачным сервером, то клиент имеет прямой доступ к его методам через указатель на виртуальную таблицу интерфейсов компонентов. На рис. 3 показано, как осуществляется транспортировка в случае локального и удаленного серверов. Подробнее транспортировка будет обсуждаться ниже в примерах этой главы.

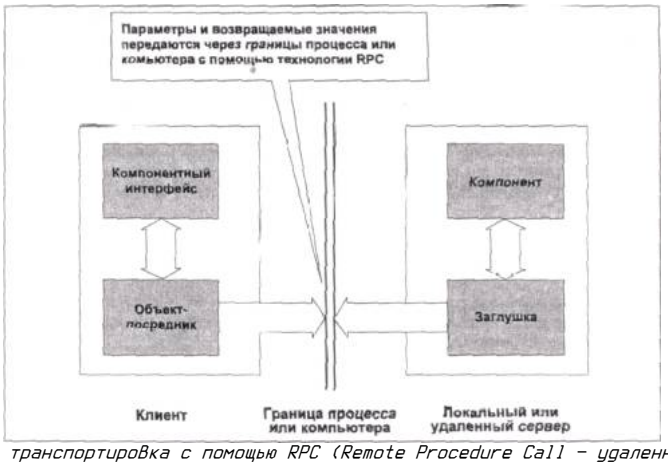

Рис. 3. Межзадачная транспортировка с помощью RPC (Remote Procedure Call – удаленный вызов процедур)

Фабрики классов

В файле, в котором хранится компонент, должны присутствовать также средства, обеспечивающие стандартный, независимый от языка способ создания экземпляров этого компонента по запросу клиента. Средства COM предоставляют стандартный интерфейс IClassFactory, который должен обеспечивать создание экземпляров компонентов по запросу извне. Ниже приводится определение интерфейса IClassFactory. Как и все COM-интерфейсы, он должен реализовывать интерфейс IUnknown.

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

Фабрика классов представляет собой COM-компонент, единственной задачей которого является облегчение создания других COM-компонентов. Всякое хранилище компонента, будь то исполняемый или DLL-файл, должно обеспечивать реализацию фабрики классов для каждого компонента, который может создаваться по внешним запросам. Основной интерфейс IClassFactory предоставляет два метода: CreateInstance, создающий экземпляр компонента, и LockServer, обеспечивающий блокировку программы сервера в памяти. Блокируя сервер для других программ, клиент получает уверенность в том, что доступ к нему будет быстро получен. Обычно это делается в целях повышения производительности.

Рассмотрим интерфейс фабрики классов для компонента Math.

```
class MathClassFactory : public IClassFactory
{
protected:
   // Счетчик обращений к экземпляру ClassFactory
   long m_lRef;
public:
   MathClassFactory() ;
   ~MathClassFactory() ;

   // реализация интерфейса IUnknown
   virtual HRESULT QueryInterface (REFIID riid, void** ppv);
   virtual ULONG AddRef();
   virtual ULONG Release();

   // реализация интерфеса  IClassFactory
   virtual HRESULT CreateInstance(LPUNKNOWN pUnk, REFIID riid, void** ppv);
   virtual HRESULT LockServer(BOOL fLock) ;
};

HRESULT MathClassFactory::CreateInstance( LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj )
{
   Math *pMath;
   HRESULT hr;
```

```
    // Инициализация возвращаемого значения
    // на случай возникновения проблем
    *ppvObj = NULL;

    // Создание нового экземпляра объекта Math
    pMath = new Math;
    if ( pMath == 0 )
       return( E_OUTOFMEMORY );

    // Получить у созданного объекта указатель
    // на запрашиваемый интерфейс
    hr = pMath->QueryInterface( riid, ppvObj ) ;
    if ( FAILED( hr ) )
       delete pMath;

    return hr;
}
```

Фабрика классов необходима для каждого компонента, а хранилище компонента должно обеспечивать для средств COM способ доступа к этой фабрике. В зависимости от варианта хранения используется одна из двух основных технологий доступа. DLL-файлы должны предоставлять в общее пользование две функции: DllGetClassObject и DllCanUnloadNow, а исполняемые файлы должны регистрировать свои фабрики классов с помощью функции CoRegisterClassObject из библиотеки COM API.

В данном случае мы реализуем компонент Math как внутризадачный сервер. В функцию DllGetClassObject включим следующую программу:

```
STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid, void** ppv )
{
   HRESULT             hr;
   MathClassFactory   *pCF;
   pCF = 0;

   // необходимо убедиться, что CLSID соответствует нашему компоненту
   if (rclsid != CLSID_Math)
      return (E_FAIL );

   pCF = new MathClassFactory;
   if ( pCF == 0 )
      return( E_OUTOFMEMORY ) ;

   hr = pCF->QueryInterface ( riid, ppv ) ;

   // Проверка возвращаемого значения QueryInterface
   if ( FAILED( hr ) )
   {
      delete pCF;
      pCF = 0;
   }

   return hr;
}
```

Не забудьте, что пользователю компонента Math потребуется указатель на интерфейс IMath. Вначале пользователь создаст экземпляр компонента, а потом запросит этот интерфейс. Средства COM предоставляют базовые функции API, позволяющие пользователю создавать экземпляры COM-компонентов. В следующей программе показано, каким образом клиент может получить доступ к компоненту Math. Из этой программы удален фрагмент, обеспечивающий выявление ошибок, благодаря чему она получилась краткой и наглядной.

```
//
// Client.cpp
//
#include "client.h"
#include <initguid.h>
#include "imath.h"
int main( int argc, char *argv[] )
{
   CoInitialize( NULL );
   HRESULT hr = CoGetClassObject( CLSID_Math,
   CLSCTX_INPROC, NULL,
   IID_IClassFactory, (void**) &pCF ) ;

   // Используя интерфейс фабрики классов, создаем экземпляр
   // компонента Math
   IUnknown* pUnk;
   hr = pCF->CreateInstance( NULL, IID_IUnknown, (void**) &pUnk );

   // Освободить интерфейс фабрики классов
   pCF->Release() ;
   IMath* pMath = NULL;
   hr = pUnk->QueryInterface( IID_IMath, (LPVOID*)&pMath );
   pUnk->Release() ;

   long result;
   result = pMath->Multiply( 100, 8 );
   cout << "100 * 8 is " << result << endl;
   result = pMath->Subtract( 1000, 333 );
   cout << "1000 - 333 is " << result << endl;
```

```
    pMath->Release () ;

    CoUninitialize();
    return 0;
}
```

Прежде чем использовать какую-либо функцию COM API, клиент должен инициализировать сервисы COM с помощью функции CoInitialize. Как только это будет сделано, клиент должен применить функцию CoGetClassObject для получения требуемого интерфейса фабрики классов компонента (заданного идентификатором CLSID компонента, который будет рассмотрен чуть ниже). Тотчас после получения клиентом фабрики классов он вызывает метод CreateInstance() для создания фактического экземпляра класса Math и освобождает интерфейс фабрики классов.

Метод CreateInstance() возвращает указатель на заданный интерфейс. В нашем примере требуется интерфейс IUnknown, и сразу же после его получения мы применяем метод QueryInterface для получения указателя на интерфейс IMath. Необходимый интерфейс IMath можно было легко получить и с помощью вызова CreateInstance, но мы хотели показать вам пример использования методов QueryInterface() и Release().

Указатель на интерфейс IMath используется для выполнения некоторых основных вычислений. После их окончания мы вызываем метод Release () и удаляем экземпляр компонента. Может возникнуть вопрос: "Откуда COM известно место хранения компонента (DLL)?" Ответим: "Из реестра". В COM для хранения информации о компонентах широко используется реестр Windows.

Реестр

Информация, необходимая сервисам COM и приложениям-клиентам для размещения и создания экземпляров компонентов, хранится в реестре Windows. Обратившись к реестру, приложения могут определить количество и тип установленных в системе компонентов и т.д.

Информация в реестре упорядочена иерархически и имеет несколько предопределенных высокоуровневых разделов. В этой главе наибольшее значение для нас будет иметь раздел HKEY_CLASSES_ROOT, в котором хранится информация о компонентах. Важным подразделом HKEY_CLASSES_ROOT является CLSID (идентификаторы классов), в котором описывается каждый компонент, установленный в системе. Например, компонент Math, который мы создали, нуждается для своей работы в нескольких элементах реестра. Перечислим их:

```
HKEY_CLASSES_ROOT\Math.Component.1 = Chapter 6 Math Component
HKEY_CLASSES_ROOT\Math.Component.1\CurVer = Math.Component.1
HKEY_CLASSES_ROOT\Math.Component.1\CLSID = {A888F560-58E4-11d0-A68A-0000837E3100}

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100} = Chapter 6 Math Component

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\ProgID = Math.Component.1

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\VersionIndependentProgID = Math.Component

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\InprocServer32 = c:\book\chap6\server\debug\server.dll

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\NotInsertable
```

В первых трех строках создается *программный идентификатор* (ProgID) для компонента Math. Идентификатор класса CLSID компонента является его уникальным идентификатором, однако он очень неудобен для чтения и запоминания. Понятие ProgID включено в COM для облегчения взаимодействия с компонентами разработчиков. Более подробно этот аспект будет рассмотрен в следующем разделе данной главы. В третьей строке обеспечивается непосредственная связь между ProgID нашего элемента управления и соответствующим CLSID.

В последних строках рассматриваемого фрагмента кода содержится вся информация, необходимая COM для помещения компонентов на хранение.

Между ProgID и информацией о версии компонента существует взаимосвязь. Однако наиболее важным элементом является раздел InProcServer32, который описывает точное положение хранилища компонентов. В таблице 2 основные элементы реестра описаны более подробно.

Элемент	Назначение
ProgID	Задает строку ProgID для COM-класса. Может содержать 39 символов, в том числе и символ точки.
InProcServer32	Содержит путь и имя 32-разрядного DLL-файла. Наличие пути необязательно, но если он не указан, то загрузка компонента возможна только в случае размещения его в одном из каталогов команд Windows (задается переменной среды PATH). Понятие *внутризадачный сервер* будет рассмотрен немного позже.
LocalServer32	Содержит путь и имя 32-битового EXE-файла. Понятие *локальный сервер* будет рассмотрен немного позже.
CurVer	ProgID последней версии класса компонента.

Таблица 2. Основные элементы реестра

Категории компонентов

Основные элементы регистра предоставляют только ограниченную информацию о компоненте. При зарождении COM-технологий несколько его элементов вполне удовлетворяли потребности в определении огромных функциональных возможностей компонента. Отсутствие или существование подраздела также несет в себе существенную информацию. Однако теперь, когда на рядовом компьютере обычно устанавливается множество COM-компонентов, к классификации их возможностей следует относиться более внимательно.

Корпорация Microsoft не оставила без внимания эту потребность в классификации и предложила новый механизм описания функций компонентов. Новая спецификация предоставила как определяемые системой, так и задаваемые пользователем категории для различных компонентов и получила название *категории компонентов*. Информация о категориях заложена в реестре, но также существует возможность вводить и удалять новые категории, даже не вникая в подробности строения реестра. Более подробно эти вопросы мы рассмотрим в следующей главе.

Утилита OLEVIEW

Хорошим средством просмотра реестра с точки зрения COM является утилита OLEVIEW, предоставляемая Visual C++ и средой SDK. Она

обеспечивает несколько различных подходов к использованию компонентов в Вашей системе, а также обладает множеством других полезных свойств. На рис. 4 показан результат использования утилиты OLEVIEW.

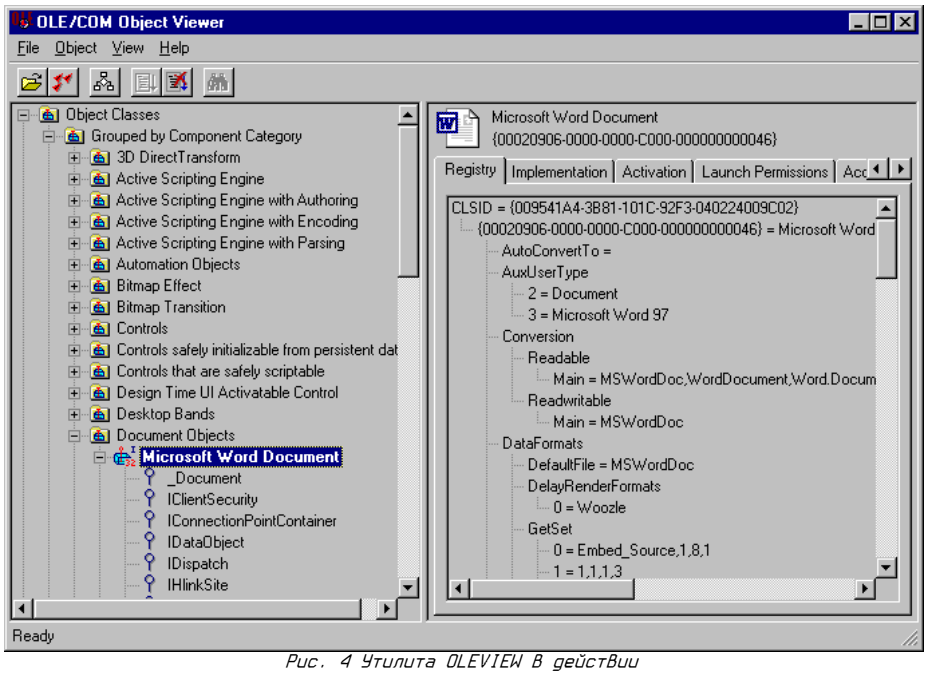

Рис. 4 Утилита OLEVIEW в действии

Глобальные уникальные идентификаторы (GUID)

В распределенных объектных или компонентных средах уникальная идентификация компонентов имеет первоочередное значение. В COM используются технологии, описанные в стандарте распределенных Вычислительных сред (Distributed Computing Environment - DCE) и используемые для вызовов удаленных процедур (Remote Procedure Call - RPC). Стандарт описывает и такое понятие, как универсальный уникальный идентификатор (Universally Unique Identifier - UUID). Реализация Win32 RPC основана на стандарте OSF RPC и, таким образом, широко использует этот механизм идентификации.

UUID представляет собой 128-разрядное значение, которому гарантируется статистическая уникальность. Это достигается путем сочетания уникального сетевого адреса (48 битов) с множеством других значений. Используемый в COM UUID называется глобальным уникальным идентификатором (Globally Unique Identifier - GUID), но в основных чертах он полностью идентичен UUID. В COM идентификаторы GUID используются для идентификации классов компонента (CLSID), интерфейсов (IID), библиотек типов и категорий компонентов (CATID). Ниже приведены GUID, используемые в нашем примере компонента Math.

```
// {A888F560-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( CLSID_Math,
             0xa888f560, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);

// {A888F561-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( IID_IMath,
             0xa888f561, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);
```

Макрос DEFINE_GUID создает глобальную константу, которая может быть использована в любой Вашей программе, как на стороне клиента, так и на стороне сервера. Однако ее значение определяется один раз. Система программирования COM предоставляет множество макросов, предназначенных для облегчения работы с глобальными идентификаторами. В той точке Вашей программы, в которой требуется определить GUID-структуру, необходимо перед файлом заголовков с объявлениями Включить файл INITGUID.H. Вот как это Выглядит на нашем примере компонента Math:

```
//
// imath.h
//
// {A888F560-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( CLSID_Math,
             0xa888f560, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);
// {A888F561-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( IID_IMath,
             0xa888f561, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);

class IMath : public IUnknown
{
public:
   virtual long Add(long Op1, long Op2) = 0;
   virtual long Subtract(long Op1, long Op2) = 0;
   virtual long Multiply(long Op1, long Op2) = 0;
   virtual long Divide (long Op1, long Op2) = 0;
};

#include "client.h"
#include <initguid.h>
#include "imath.h"
```

За счёт Включения в программу файла INITGUID.H мы отменяем значение макроса DEFINE_GUID таким образом, что он не просто объявляет тип переменной GUID, а создает и инициализирует переменную этого типа.

В примере Math нам потребовались два GUID. Идентификатор CLSID идентифицирует сам компонент, а IID – пользовательский COM-интерфейс. Существует множество способов генерации GUID для компонентов.

Если Вы пользуетесь утилитами среды Visual C++ AppWizard и ClassWizard или мастером объектов ATL, то GUID будут генерироваться автоматически. Программно создать их можно с помощью COM-функции CoCreateGuid или двух программ Visual C++. Одна из них, UUIDGEN, вызывается из командной строки, и ее можно применить при создании последовательности GUID для целого проекта. Приведенное ниже содержимое командной строки обеспечит создание списка из 50 GUID и запись их в заданный файл.

```
c:\msdev\bin\uuidgen -n50 > Project_Guids.txt
```

Другая программа, GUIDGEN, открывает собственное окно и обеспечивает несколько способов представления созданных GUID. В нашем случае требуется формат DEFINE_GUID. Используя кнопку Copy, можно вставить определение GUID через буфер обмена непосредственно в исходный текст программы. На рис. 5 показано окно программы GUIDGEN.

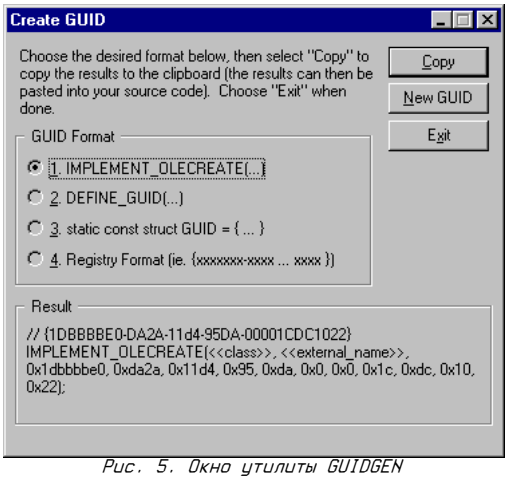

Рис. 5. Окно утилиты GUIDGEN

Библиотека COM API предоставляет несколько функций, предназначенных для сравнения, создания и преобразования типов GUID. Наиболее употребительные из них приведены в табл. 3.

Функция	Назначение
CoCreateGuid(GUID* pGuid)	Обеспечивает создание одного уникального GUID
IsEqualGUID(REFGUID, REFGUID)	Сравнивает два GUID
IsEqualIID (REFIID, REFIID)	Сравнивает два IID
IsEqualCLSID(REFCLSID, REFCLSID)	Сравнивает два CLSID
CLSIDFromProgID(LPCOLESTR, LPCLSID)	Возвращает CLSID для заданного ProgID
ProgIDFromCLSID(REFCLSID, LPOLESTR*)	Возвращает ProgID для заданного CLSID

Таблица 3 Вспомогательные функции GUID

Программные идентификаторы

Компонент однозначно идентифицируется своим CLSID. Однако запоминание CLSID многочисленных компонентов может представлять затруднения. Поэтому предлагается другой механизм именования компонентов, а именно программный идентификатор, или ProgID, который представляет собой простую символьную строку, связываемую с определенным компонентом через реестр. Например, пусть для компонента Math мы выбрали ProgID равный Math.Component. С помощью ProgID определение удобного для понимания имени компонента намного упрощается, как, например, в этом фрагменте программы, написанной на Visual Basic:

```
Dim objMath As Object
Set objMath = CreateObject("Math.Component")
objMath.Add(100, 100)
Set objMath = Nothing
```

Функция Visual Basic CreateObject в качестве параметра принимает ProgID компонента. На самом деле для преобразования ProgID в действительный CLSID компонента оператор использует COM-функцию CLSIDFromProgID. А затем уже CreateObject создает экземпляр компонента с помощью функции CoCreateInstance.

Функции COM API

Фирма Microsoft поставляет множество функций Win32 API, специально предназначенных для COM, ActiveX и OLE. Существует более сотни функций, предназначенных для работы с COM, поэтому мы не в состоянии здесь рассмотреть их все. Однако, изучив основные API-функции COM, мы получим хорошее представление о работе COM. API-функции COM обеспечивают основу высокоуровневых сервисов, таких как OLE и ActiveX. Нужно также не забывать, что COM – это только группа определений интерфейсов, которые должны быть реализованы пользователем, а вызовы API всего лишь дают возможность сделать это. В табл. 4 приведены API-функции, которые будут упомянуты в этой главе.

Функция	Назначение
CoInitialize CoInitializeEx (Клиент и сервер)	Инициализирует COM-библиотеки для использования их в программе.
CoUninitialize (Клиент и сервер)	Освобождает COM-библиотеки при отсутствии потребности в них. Не используется внутризадачными серверами.

CoGetClassObject (Клиент)	Обращение к фабрике классов для получения экземпляра заданного COM-объекта.
CoCreateGuid (Клиент и сервер)	Создает новый уникальный GUID.
CoCreateInstance (Клиент) CoCreateInstanceEx (Клиент)	Создаст новый экземпляр COM-объекта (Возможно, на удаленном компьютере).
CoRegisterClass (EXE-сервер)	Регистрирует фабрику классов для определенного COM объекта.
DllCanUnloadNow (Внутризадачный сервер)	Периодически вызывается COM для проверки возможности выгрузки DLL (проверяет отсутствие экземпляров объектов, хранящихся в DLL). Реализуется во внутризадачном сервере.
DllGetClassObject (Внутризадачный сервер)	Точка входа, реализуемая во внутризадачных серверах так, что интерфейсы их фабрик классов могут быть получены клиентами.

Таблица 4. Основные функции COM

CoInitialize и CoInitializeEx

Функция CoInitialize инициализирует статические и загружаемые библиотеки COM, после чего могут использоваться остальные функции COM API. Эта функция принимает один параметр, зарезервированный для использования в будущих версиях COM, поэтому его значение должно быть равно null. Функция CoInitializeEx была добавлена для поддержки различных потоковых моделей COM. До создания Windows NT 4.0 COM поддерживала только потоковую модель с раздельным адресным пространством, которая принималась по умолчанию. Функция CoInitializeEx имеет два параметра. Первый из них зарезервирован и всегда должен иметь значение null, а второй определяет одну из потоковых моделей, заданных в перечислении COINIT.

```
typedef enum tagCOINIT
{
   COINIT_APARTMENTTHREADED  = 0x2, // Использует раздельное адресное пространство
                                    // потоков

#if  (_WIN32_WINNT >= 0x0400 ) || defined(_WIN32_DCOM) // DCOM
   // These constants are only valid on Windows NT 4.0
   COINIT_MULTITHREADED      = 0x0, // OLE вызывает объекты из любого потока.
   COINIT_DISABLE_OLE1DDE    = 0x4, // Не использует DDE для поддержки OLE1.
   COINIT_SPEED_OVER_MEMORY  = 0x8, // Для увеличения скорости занимает
                                    // больше памяти.
#endif // DCOM
} COINIT;
```

CoUninitialize

Функция CoUninitialize вызывается, если требуется освободить ресурсы статических и загружаемых библиотек COM. Вызов возможен только в том случае, если перед этим произошел успешный вызов функции CoInitialize. С другой стороны, после каждого вызова функции CoInitialize необходимо вызывать функцию CoUninitialize.

CoRegisterClassObject

Функция CoRegisterClassObject вызывается сервером для регистрации фабрик классов как доступных к использованию. Эта функция должна вызываться для каждой фабрики классов, содержащейся в исполняемом коде, причем как можно раньше, желательно до начала цикла обработки сообщений Windows. Функция CoRegisterClassObject может быть использована только в исполняемых файлах (EXE). Внутризадачные серверы для доступа к их фабрикам классов используют функцию DllGetObject (табл. 5).

Параметр	Описание
REFCLSID rclsid	CLSID для зарегистрированного класса компонента.
LPUNKNOWN pUnk	Указатель на интерфейс IUnknown для зарегистрированного класса компонента.
DWORD dwClsContext	Запрашиваемый контекст для выполняемого кода.Может принимать следующие значения: CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER.
DWORD flags	Флаги REGCLS определяют способ создания нескольких экземпляров компонента. Переменная flags может принимать следующие значения: REGCLS_SINGLEUSE REGCLS_MULTIPLEUSE REGCLS_MULTI_SEPARATE.
LPDWORD lpdwRegister	Возвращаемое значение, используется при отмене регистрации объекта класса с помощью функции CoRevokeClassObject.

Таблица 5. Параметры функции CoRegisterClassObject

CoGetClassObject

Функция CoGetClassObject используется клиентом COM для получения указателя на интерфейс IClassFactory заданного класса компонента. Затем этот указатель может быть использован для создания нескольких экземпляров компонента.

COM определит вид сервера – загружаемый (DLL-файл) или выполняемый (EXE-файл). Если компонент хранится в DLL-файле, то COM загрузит его и передаст клиенту указатель на требуемый интерфейс с помощью вызова своей функции DllGetClassObject. Если же компонент содержится в исполняемом файле, который в данный момент не запущен, то COM запустит его локально или удаленно, подождет, пока сервер зарегистрирует свои фабрики классов с помощью функции CoRegisterClassObject, а затем возвратит клиенту

требуемый интерфейс.

В NT Версии 4.0 и Выше параметр *coserverinfo* используется для разрешения создания экземпляра на удаленных системах. Структура *coserverinfo* позволяет задать имя сервера как имя UNC (например, \\twa_nt), имя DNS (например, www.WidgetWare.com) или IP-адрес (например, 191.51.33.1).

```
typedef struct  _COSERVERINFO
    {
    DWORD dwReserved1;
    LPWSTR pwszName;
    COAUTHINFO __RPC_FAR *pAuthInfo;
    DWORD dwReserved2;
    }    COSERVERINFO;
```

В большинстве случаев клиент должен применять сокращенный вариант функции CoCreateInstance, описываемой далее. Приложение клиента может использовать функцию CoGetClassObject в двух случаях. Во-первых, если в этом приложении необходимо создать несколько экземпляров компонентного объекта. Тогда получение только одной копии фабрики классов для их создания будет более эффективным. Во-вторых, если клиенту требуется получить доступ к методу IClassFactory::LockServer () с целью блокировки в силу каких-либо причин хранилища компонента в памяти. Обычно это опять-таки связано с повышением эффективности. В табл. 6 приведены параметры функции CoGetClassObject.

Параметр	Описание
REFCLSID rclsid	Ссылка на CLSID для заданного компонента.
DWORD dwClsContext	Запрашиваемый контекст для хранилища сервера. Может принимать одно или несколько из следующих значений: CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER.
COSERVERINFO pServerInfo	Указатель на структуру COSERVERINFO.
REFIID riid	Ссылка на IID для интерфейса, который необходимо возвратить из созданного объекта класса. Обычно это IClassFactory. Благодаря этому клиент может создать экземпляр требуемого компонента.
VOID** ppvObj	Указатель типа void* на интерфейс, подлежащий возврату

Таблица 6. Параметры функции CoGetClassObject

CoCreateInstance

Функция CoCreateInstance используется приложением клиента для создания экземпляра заданного класса компонента. Существует вспомогательная функция, называемая CoGetClassObject, предназначенная для получения фабрики классов компонента и последующего использования метода IClassFactory::CreateInstance() с целью создания экземпляра компонента. Однако вместо выполнения приведенного ниже трех шагового процесса для получения необходимого Вам интерфейса лучше использовать функцию CoCreateInstance.

```
// Функция CoCreateInstance Выполняет такие действия:
CoGetClassObject(..., &pCF) ;
pCF->CreateInstance(..., &pInt);
pCF->Release();
```

Параметры функции CoCreateInstance аналогичны параметрам функции CoGetClassObject. Единственное отличие состоит в том, что использование клиентом функции CoCreateInstance приведет к запросу заданного Вами интерфейса для компонента (например, IDispatch) вместо указателя на IClassFactory.

Осуществляя поддержку распределенной модели COM, функция CoGetClassObject использует ранее зарезервированный параметр для передачи структуры COSERVERINFO. Однако функция CoCreateInstance не имела такого зарезервированного параметра. Поэтому потребовалась новая функция, CoCreateInstanceEx, которая применяется для создания COM-объекта на удаленном компьютере. Четвертый параметр этой функции как раз и используется для передачи структуры COSERVERINFO. Формат вызова функции CoCreateInstanceEx аналогичен описанному ранее для вызова функции CoGetClassObject.

В целях повышения производительности при создании экземпляра компонента имеется структура MULTI_QI, задачей которой является разрешение доступа клиенту к QueryInterface для получения нескольких интерфейсов за один вызов. Данная структура позволяет организовывать массив IID, который будет возвращаться вместе с массивом указателей на интерфейсы.

```
typedef struct  tagMULTI_QI
    {
    const IID __RPC_FAR *pIID;
    IUnknown __RPC_FAR *pItf;
    HRESULT hr;
    }    MULTI_QI;
```

Параметры функций CoCreateInstance и CoCreateInstanceEx приведены в табл. 7 и 8.

Параметр	Описание
REFCLSID rclsid	Ссылка на CLSID для заданного компонента.
IUnknown* pUnkOuter	При использовании механизма агрегации параметры управляющей среды неизвестны.
DWORD dwClsContext	Запрашиваемый контекст для хранилища сервера. Может принимать одно, два или все из следующих значений: CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER

	CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER.
REFIID riid	Ссылка на IID для заданного интерфейса, который необходимо Возвратить из созданного компонентного объекта.
VOID** ppvObj	Указатель типа void* на Возвращаемый интерфейс.

Таблица 7. Параметры функции CoCreateInstance

Параметр	Описание
REFCLSID rclsid	Ссылка на CLSID для заданного компонента.
IUnknown* pUnkOuter	При использовании механизма агрегации параметры управляющей среды неизвестны.
DWORD dwClsContext	Запрашиваемый контекст для хранилища сервера. Может принимать одно, два или все из следующих значений: CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER.
COSERVERINFO* pServerInfo	Указатель на структуру COSERVERINFO.
ULONG	Количество вызовов QueryInterface, которые необходимо осуществить для обработки структуры MULTI_QI.
MULTI_QI	Массив структур MULTI_QI. Получение сразу множества интерфейсов с помощью одного вызова функции более эффективно.

Таблица 8. Параметры функции CoCreateInstanceEx

DllCanUnloadNow

Функция DllCanUnloadNow используется во внутризадачных серверах. С ее помощью COM периодически проверяет, можно ли произвести выгрузку DLL-файла. Функция не имеет параметров и возвращает либо значение S_FALSE, указывающее, что DLL-файл не может быть выгружен, либо S_OK, обозначающее обратное.

DllGetClassObject

Функция DllGetClassObject используется во внутризадачных серверах для обращения к фабрикам классов компонентных объектов. При запросе приложением клиента компонента, хранящегося во внутризадачном сервере, COM вызывает точку входа DLL-файла – функцию DllGetClassObject – с параметрами, описанными в табл. 9.

Параметр	Описание
REFCLSID rclsid	Ссылка на CLSID для заданного компонента.
DWORD dwClsContext	Запрашиваемый контекст для хранилища сервера. Может принимать одно из следующих значений: CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER
LPVOID pvReserved	Зарезервирован. Должен иметь значение NULL.
REFIID riid	Ссылка на IID для интерфейса, который необходимо Возвратить из созданного компонентного объекта. Обычно это IClassFactory, и таким образом клиент получает возможность создать экземпляр необходимого компонента.
VOID** ppvObj	Указатель типа void* на Возвращаемый интерфейс.

Таблица 9. Параметры функции CoGetClassObject

Часть 4. «Говорящий не знает, знающий не говорит. Лао Цзы» – от теории к практике!

Простые COM-клиенты и COM-серверы

Для иллюстрации описанных приемов работы с COM приведем два примера на языке C++, в которых используется COM. Они достаточно просты и используют язык C++ и функции COM API. Всю работу будем проделывать самостоятельно, без использования таких структур, как MFC или ATL. Хотя примеры и тривиальны, на них вполне можно продемонстрировать основные приемы, используемые в COM. Ниже в этой главе мы применим новую библиотеку активных шаблонов фирмы Microsoft, чтобы заново реализовать данный пример сервера.

Прежде чем мы начнем разработку, необходимо внести существенное изменение в интерфейс компонента. Первоначальное определение абстрактного класса выглядит следующим образом:

```
class IMath : public IUnknown
{
public:
   virtual long Add(long Op1, long Op2) = 0;
   virtual long Subtract(long Op1, long Op2) = 0;
   virtual long Multiply(long Op1, long Op2) = 0;
   virtual long Divide (long Op1, long Op2) = 0;
};
```

Такой способ объявления порождает одну проблему. Каждый метод COM-интерфейса должен возвращать значение типа HRESULT. В нашем случае возвращается только результат операции. Поэтому нам требуется возвратить HRESULT, а результат операции передать через параметр. Это выглядит следующим образом:

```
class IMath : public IUnknown
{
public:
   virtual HRESULT Add(long Op1, long Op2, long *pResult) = 0;
   virtual HRESULT Subtract(long Op1, long Op2, long *pResult) = 0;
   virtual HRESULT Multiply(long Op1, long Op2, long *pResult) = 0;
   virtual HRESULT Divide (long Op1, long Op2, long *pResult) = 0;
};
```

Такая запись несколько необычна, поскольку известно, что возврат методом значения вычислений уменьшает сложность программы. Теперь же приходится работать с указателем на возвращаемый результат. Возвращение значения типа HRESULT каждым методом является общим правилом в COM. Однако существуют технические приемы (например, ключевое слово языка определения интерфейсов retval), позволяющее приложению клиента трактовать каждый метод таким образом, как если бы он на самом деле возвращал результат, а не HRESULT. Ниже мы рассмотрим эту возможность на примере клиента.

Макросы STDMETHOD и STDMETHODIMP

Файлы заголовков COM предоставляют несколько макросов, используемых при объявлении и реализации COM-интерфейсов. Из соображений простоты вплоть до этого момента мы использовали в примерах обыкновенную C++-программу. Однако Microsoft рекомендует применять именно эти макросы, поскольку благодаря им исключается возможность несовместимости программных сред. В первую очередь разберемся с макросами **STDMETHOD** и **STDMETHOD_**, а также с **STDMETHODIMP** и **STDMETHODIMP_**. Вот новое определение интерфейса IMath с использованием макроса **STDMETHOD**.

```
class IMath : public IUnknown
{
public:
   STDMETHOD( Add(long , long , long *))PURE;
   STDMETHOD( Subtract(long , long , long *))PURE;
   STDMETHOD( Multiply(long , long , long *))PURE;
   STDMETHOD( Divide (long , long , long *))PURE;
};
```

Результат развертывания макроса **STDMETHOD_** зависит от целевой среды и языка программирования: C или C++. Для среды Win32 с использованием C++ определение этого макроса выглядит следующим образом:

```
// OBJBASE.H
#define STDMETHODCALLTYPE       __stdcall
#define STDMETHOD(method)       virtual HRESULT STDMETHODCALLTYPE method
#define STDMETHOD_(type,method) virtual type STDMETHODCALLTYPE method
#define PURE                    = 0
#define STDMETHODIMP            HRESULT STDMETHODCALLTYPE
#define STDMETHODIMP_(type)     type STDMETHODCALLTYPE
```

Если развернуть этот макрос, то наша программа станет очень похожа на предыдущие примеры. Единственное отличие состоит в добавлении модификатора __stdcall. Этот модификатор указывает на необходимость соблюдения определенного соглашения фирмы Microsoft относительно вызовов, используемого в API-функциях Win32. Это соглашение требует, чтобы вызываемая программа очищала стек после своего вызова. Модификатор **PURE** – всего лишь способ обозначить функцию как чисто виртуальную (pure virtual, т.е. = 0).

Большая часть методов COM-интерфейсов возвращает стандартный тип HRESULT. В этом состоит единственное различие между макросами **STDMETHOD** и **STDMETHOD_**, поскольку метод, определенный как **STDMETHOD**, всегда возвращает данные типа HRESULT, а определение макроса **STDMETHOD_** позволяет пользователю задавать тип возвращаемого значения. Вот как в определении интерфейса IClassFactory используется макрос **STDMETHOD**:

```
STDMETHOD (LockServer(BOOL fLock)) PURE;
//Разворачивается таким образом
virtual HRESULT _stdcall LockServer(BOOL fLock)= 0;
```

Макрос **STDMETHOD** применен для объявления методов интерфейса, как в абстрактных определениях, так и в определениях класса. Единственным отличием является модификатор PURE. Приведем программу для производного класса:

```
class Math: public IMath
{
   ...
public:
   //IUnknown
   STDMETHOD(QueryInterface( REFIID, void** ));
   STDMETHOD_(ULONG, AddRef());
   STDMETHOD_(ULONG,Release());

   //IMath
   STDMETHOD(Add( long, long, long*));
   STDMETHOD(Substract( long, long, long*));
   STDMETHOD(Multiply( long, long, long*));
   STDMETHOD(Divide( long, long, long*));
};
```

И, наконец, при реализации класса используется макрос STDMETHODIMP. Приведем пример релизации класса Math:

```
STDMETHODIMP Math::Add( long Op1, long Op2, long *pResult )
{
   *pResult = Op1 + Op2;
   return S_OK;
```

```
}

STDMETHODIMP Math::Subtract( long Op1, long Op2, long *pResult )
{
   *pResult = Op1 - Op2;
   return S_OK;
)

STDMETHODIMP Math:: Multiply( long Op1, long Op2,long *pResult )
{
   *pResult = Op1 * Op2;
   return S_OK;
}

STDMETHODIMP_ (long) Math::Divide( long Op1, long Op2, long *pResult )
{
   *pResult = Op1 / Op2;
   return S_OK;
}
```

Проект сервера

Приложение сервера обеспечивает реализацию интерфейса IMath и, таким образом, создает и помещает на хранение компонент Math. Его определение встречалось на протяжении всей данной главы, поэтому здесь мы не будем останавливаться на нем подробно. Для разработки примера сервера воспользуемся средой разработки Visual C++. Затем с помощью утилиты AppWizard создадим проект Dynamic Link Library с именем Server. Таким образом, мы получим простой проект Visual C++, не имеющий исходных файлов. На рисунке 6. изображено диалоговое окно, с помощью которого создается новый проект.

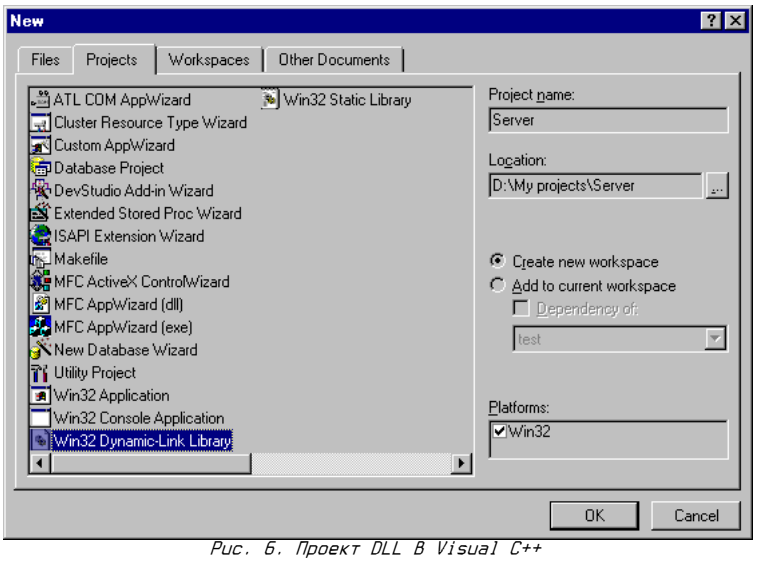

Рис. 6. Проект DLL в Visual C++

Теперь необходимо объявить интерфейс абстрактного компонента и его CLSID и IID. Ранее это уже делалось, но сейчас все фрагменты кода собраны вместе. Наберите следующий текст и сохраните его в файле с именем IMATH.H.

```
//
// imath.h
//

// {A888F560-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( CLSID_Math,
             0xa888f560, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);

// {A888F561-58E4-11d0-A68A-0000837E3100}
DEFINE_GUID( IID_IMath,
             0xa888f561, 0x58e4, 0x11d0, 0xa6, 0x8a, 0x0, 0x0, 0x83, 0x7e, 0x31, 0x0);

class IMath : public IUnknown
{
public:
   STDMETHOD( Add( long, long, long* ))      PURE;
   STDMETHOD( Subtract( long, long, long* )) PURE;
   STDMETHOD( Multiply( long, long, long* )) PURE;
   STDMETHOD( Divide( long, long, long* ))   PURE;
};
```

Чтобы предоставить программе клиента информацию об определениях интерфейса и идентификаторов CLSID и IID, необходимо отделить их от собственно текста программы. Только благодаря этому программа клиента может получить доступ к функциям компонента. Фактически клиент не нуждается в CLSID (поскольку доступ к компоненту осуществляется с помощью его ProgID), поэтому ничто не помешает нам расположить его именно здесь.

Строки, начинающиеся с макроса DEFINE_GUID, можно ввести таким же образом, как это сделано здесь, либо создать ваши собственные с помощью утилиты GUIDGEN. Теперь необходимо объявить класс компонента и фабрику классов для него. Чтобы сделать это, создайте файл с именем MATH.H и введите следующие определения:

```
//
```

```
// math.h
//

#include "imath.h"

extern long g_lObjs;
extern long g_lLocks;


class Math : public IMath
{
protected:
   // Reference count
   long          m_lRef;

public:
   Math();
   ~Math();

public:
   // IUnknown
   STDMETHOD(QueryInterface( REFIID, void** ));
   STDMETHOD_(ULONG, AddRef());
   STDMETHOD_(ULONG, Release());

   // IMath
   STDMETHOD(Add( long, long, long* ));
   STDMETHOD(Subtract( long, long, long* ));
   STDMETHOD(Multiply( long, long, long* ));
   STDMETHOD(Divide( long, long, long* ));
};

class MathClassFactory : public IClassFactory
{
protected:
   long       m_lRef;

public:
   MathClassFactory();
   ~MathClassFactory();

   // IUnknown
   STDMETHOD( QueryInterface(REFIID, void** ));
   STDMETHOD_(ULONG, AddRef());
   STDMETHOD_(ULONG, Release());

   // IClassFactory
   STDMETHOD( CreateInstance(LPUNKNOWN, REFIID, void**));
   STDMETHOD( LockServer(BOOL));
};
```

Большая часть этих строк встречалась вам и раньше. Класс Math является производным по отношению к классу интерфейсов IMath, который в свою очередь является производным для IUnknown. Объявим методы IUnknown и IMath. Отслеживание общего количества экземпляров компонента в DLL-файле и количества вызовов IClassFactory::LockServer возлагается на две глобальные переменные. Затем объявляем класс для фабрики классов компонента Math. Теперь можно взглянуть, на текст программы. Создайте файл MATH.CPP и введите в него следующее:

```
//
// Math.cpp
//

#include <windows.h>
#include "math.h"

//
// Math class implementation
//
// Constructors
Math::Math()
{
   m_lRef = 0;

   // Увеличить значение внешнего счетчика объектов
   InterlockedIncrement( &g_lObjs );
}

// The destructor
Math::~Math()
{
   // Уменьшить значение внешнего счетчика объектов
   InterlockedDecrement( &g_lObjs );
}
```

В конструкторе внутренний счетчик инициализируется значением нуль, а значение счетчика экземпляров для DLL-файла увеличивается. Затем деструктор его уменьшает. Теперь добавьте в программу следующее:

```
STDMETHODIMP Math::QueryInterface( REFIID riid, void** ppv )
```

```
{
    *ppv = 0;

    if ( riid == IID_IUnknown || riid == IID_IMath )
        *ppv = this;

    if ( *ppv )
    {
        AddRef();
        return( S_OK );
    }
    return (E_NOINTERFACE);
}

STDMETHODIMP_(ULONG) Math::AddRef()
{
   return InterlockedIncrement( &m_lRef );
}

STDMETHODIMP_(ULONG) Math::Release()
{
   if ( InterlockedDecrement( &m_lRef ) == 0 )
   {
      delete this;
      return 0;
   }

   return m_lRef;
}
```

Таким образом, будет обеспечена реализация трех методов интерфейса IUnknown. Наш компонент поддерживает только два интерфейса: обязательный IUnknown и пользовательский IMath. Функция QueryInterface проверяет возможность получения клиентом требуемого интерфейса, а также возвращает указатель на указатель виртуальной таблицы компонента. Перед возвратом указателя увеличивается значение внутреннего счетчика обращений. Это осуществляется с помощью вызова метода AddRef(). Реализации функций AddRef() и Release() ничем не отличаются от использованных ранее.

```
STDMETHODIMP Math::Add( long lOp1, long lOp2, long* pResult )
{
   *pResult = lOp1 + lOp2;
   return S_OK;
}

STDMETHODIMP Math::Subtract( long lOp1, long lOp2, long* pResult )
{
   *pResult = lOp1 - lOp2;
   return S_OK;
}

STDMETHODIMP Math::Multiply( long lOp1, long lOp2, long* pResult )
{
   *pResult = lOp1 * lOp2;
   return S_OK;
}

STDMETHODIMP Math::Divide( long lOp1, long lOp2, long* pResult )
{
   *pResult = lOp1 / lOp2;
   return S_OK;
}
```

Мы получили текст относительно простой программы, но зато она поможет нам быстро прогрессировать в понимании COM. Следующий текст программы представляет собой реализацию класса C++ для фабрики классов.

```
MathClassFactory::MathClassFactory()
{
   m_lRef = 0;
}

MathClassFactory::~MathClassFactory()
{
}

STDMETHODIMP MathClassFactory::QueryInterface( REFIID riid, void** ppv )
{
   *ppv = 0;

   if ( riid == IID_IUnknown || riid == IID_IClassFactory )
      *ppv = this;

   if ( *ppv )
   {
      AddRef();
      return S_OK;
   }

   return(E_NOINTERFACE);
}
```

```
STDMETHODIMP_(ULONG) MathClassFactory::AddRef()
{
   return InterlockedIncrement( &m_lRef );
}

STDMETHODIMP_(ULONG) MathClassFactory::Release()
{
   if ( InterlockedDecrement( &m_lRef ) == 0 )
   {
      delete this;
      return 0;
   }

   return m_lRef;
}

STDMETHODIMP MathClassFactory::CreateInstance
     ( LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj )
{
   Math*      pMath;
   HRESULT    hr;

   *ppvObj = 0;

   pMath = new Math;

   if ( pMath == 0 )
      return( E_OUTOFMEMORY );

   hr = pMath->QueryInterface( riid, ppvObj );

   if ( FAILED( hr ) )
      delete pMath;

   return hr;
}

STDMETHODIMP MathClassFactory::LockServer( BOOL fLock )
{
   if ( fLock )
      InterlockedIncrement( &g_lLocks );
   else
      InterlockedDecrement( &g_lLocks );

    return S_OK;
}
```

Большую часть из приведенного выше кода мы уже встречали. Единственным исключением является исходный текст процедуры LockServer() . Хранилище сервера (DLL-файл) содержит переменную для подсчета обращений, обеспечивающую, при необходимости, блокировку сервера. Дальше будет показано, как именно используется этот счетчик.

После сохранения предыдущего файла MATH.CPP создайте новый и назовите его SERVER.CPP. Этот файл будет содержать главную программу реализации хранилища нашего компонента. Файлы IMATH.H, MATH.H и MATH.CPP хранят тексты программ компонента. Теперь приведем код, с помощью которого наш компонент помещается в хранилище.

```
//
// server.cpp : Defines the initialization routines for the DLL.
//

#include <windows.h>

#include <initguid.h>
#include "math.h"

long    g_lObjs = 0;
long    g_lLocks = 0;

STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid, void** ppv )
{
   HRESULT             hr;
   MathClassFactory    *pCF;

   pCF = 0;

   // Make sure the CLSID is for our Expression component
   if ( rclsid != CLSID_Math )
      return( E_FAIL );

   pCF = new MathClassFactory;

   if ( pCF == 0 )
      return( E_OUTOFMEMORY );

   hr = pCF->QueryInterface( riid, ppv );

   // Check for failure of QueryInterface
   if ( FAILED( hr ) )
   {
```

```
        delete pCF;
        pCF = 0;
    }

    return hr;
}

STDAPI DllCanUnloadNow(void)
{
    if ( g_lObjs || g_lLocks )
        return( S_FALSE );
    else
        return( S_OK );
}
```

Вначале мы включили в программу файл заголовков INITGUID.H, чтобы определить GUID, используемый в DLL-файле. Затем определили две глобальные переменные, отвечающие за подсчет обращений к хранилищу компонента. Имейте в виду: для того чтобы DLL-файл стал настоящим хранилищем компонентов, стандарт COM требует наличия в нем как минимум двух функций (на самом деле их четыре, но остальные две будут рассмотрены в следующих примерах). Сначала реализуем функцию DllGetClassObject. COM вызывает эту точку входа по требованию клиента компонента. Указанная функция проверяет, поддерживается ли затребованный клиентом компонент DLL-файлом. Удостоверившись в этом, мы создаем экземпляр фабрики классов для объекта Math и вызываем функцию QueryInterface из интерфейса, затребованного клиентом. Фабрика классов объекта Math поддерживает только интерфейсы IUnknown и IClassFactory. Если клиент или COM требует какой-либо другой интерфейс, то возвращается код ошибки. Благодаря двум глобальным переменным реализация функции DllCanUnloadNow намного упростилась. Проверяем, имеются ли ожидающие обработки экземпляры компонента Math, и подсчитываем количество вызовов функции LockServer. Если какая-либо из проверок дает положительный результат, DLL-файл не может быть выгружен.

Остался всего один шаг. Чтобы сделать обе функции, определенные в файле SERVER.CPP, доступными для общего пользования, требуется создать файл определений SERVER. DEF и ввести в него следующие строки:

```
;
; Server.def : Declares the module parameters for the DLL.
;

LIBRARY      "SERVER"
DESCRIPTION  'SERVER Windows Dynamic Link Library'

EXPORTS
    ; Имена точек входа для внешнего пользования помещаются здесь
    DllGetClassObject   PRIVATE
    DllCanUnloadNow     PRIVATE
```

Прежде чем создавать проект, используйте элемент Files into project... меню Insert для включения в проект файлов MATH.CPP, SERVER.CPP и SERVER. DEF, и лишь после этого приступайте к созданию проекта. Последним шагом будет регистрация компонента Math. К этой статье прилагается файл SERVER.REG, который выглядит следующим образом:

```
REGEDIT
HKEY_CLASSES_ROOT\Math.Component.1 = Chapter 6 Math Component
HKEY_CLASSES_ROOT\Math.Component.1\CurVer = Math.Component.1
HKEY_CLASSES_ROOT\Math.Component.1\CLSID = {A888F560-58E4-11d0-A68A-0000837E3100}

HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100} = Chapter 6 Math Component
HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\ProgID = Math.Component.1
HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\VersionIndependentProgID = Math.Component
HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\InprocServer32 = c:\book\chap6\server\debug\server.dll
HKEY_CLASSES_ROOT\CLSID\{A888F560-58E4-11d0-A68A-0000837E3100}\NotInsertable
```

Если вы использовали в примере имеющиеся GUID, то вам потребуется изменить только информацию о расположении SERVER.DLL в файле SERVER.REG в разделе InProcServer32. Однако если были сгенерированы собственные GUID, необходимо во всех строках, содержащих CLSID, обновить значения GUID. После того как вы введете необходимую информацию или обновите файл SERVER.REG, включите, его в реестр с помощью утилиты REGEDIT или дважды щелкните на названии этого файла в окне Windows Explorer.

> **ПРИМЕЧАНИЕ**
> В Windows95 и Windows NT существуют программы редактирования реестра REGEDIT.EXE и REGEDIT32.EXE. Зарегистрировать REG-файл можно, дважды щелкнув на его пиктограмме в окне Windows Explorer. Можно также ввести в командной строке start server.reg.

Теперь, после создания простого компонента Math, требуется организовать доступ к нему из приложения клиента, чтобы проверить, как он функционирует.

Приложение клиента

Клиентское приложение представляет собой простое консольное приложение (Console Application) Win32. С помощью утилиты AppWizard создайте консольное приложение с именем Client. Это опять-таки только основа проекта, и AppWizard предоставит только MAK-файл. Начальные характеристики проекта показаны на рис. 7.

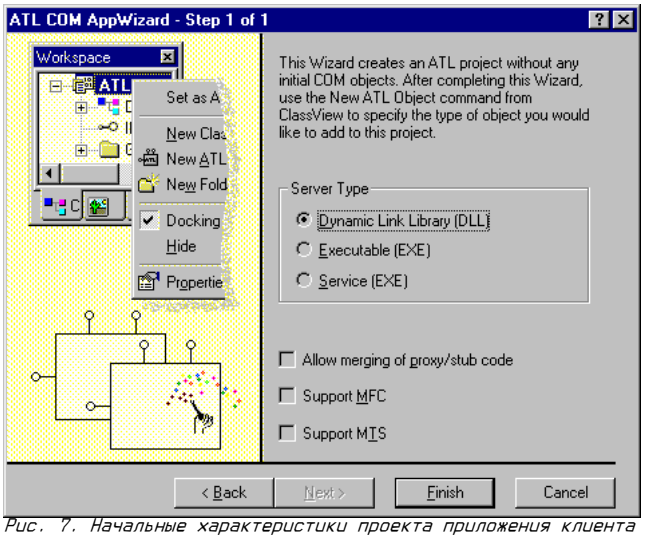

Рис. 7. Начальные характеристики проекта приложения клиента

После этого мы создадим файл CLIENT.CPP и Включим В него следующую программу:

```
//
// Client.cpp
//

#include <windows.h>
#include <tchar.h>
#include <iostream.h>

#include <initguid.h>
#include "..\server\imath.h"


int main( int argc, char *argv[] )
{
   cout << "Initializing COM" << endl;

   if ( FAILED( CoInitialize( NULL )))
   {
      cout << "Unable to initialize COM" << endl;
      return -1;
   }

   char* szProgID = "Math.Component.1";
   WCHAR  szWideProgID[128];
   CLSID  clsid;
   long lLen = MultiByteToWideChar( CP_ACP,
                       0,
                       szProgID,
                       strlen( szProgID ),
                       szWideProgID,
                       sizeof( szWideProgID ) );

   szWideProgID[ lLen ] = '\0';
   HRESULT hr = ::CLSIDFromProgID( szWideProgID, &clsid );
   if ( FAILED( hr ))
   {
      cout.setf( ios::hex, ios::basefield );
      cout << "Unable to get CLSID from ProgID. HR = " << hr << endl;
      return -1;
   }

   IClassFactory* pCF;
   // Получить фабрику классов для класса Math
   hr = CoGetClassObject( clsid,
                          CLSCTX_INPROC,
                          NULL,
                          IID_IClassFactory,
                          (void**) &pCF );
   if ( FAILED( hr ))
   {
      cout.setf( ios::hex, ios::basefield );
      cout << "Failed to GetClassObject server instance. HR = " << hr << endl;
      return -1;
   }

   // с помощью фабрики классов создать экземпляр
   // компонента и получить интерфейс IUnknown.
   IUnknown* pUnk;
   hr = pCF->CreateInstance( NULL, IID_IUnknown, (void**) &pUnk );

   // Release the class factory
   pCF->Release();
```

```
    if ( FAILED( hr ))
    {
       cout.setf( ios::hex, ios::basefield );
       cout << "Failed to create server instance. HR = " << hr << endl;
       return -1;
    }

    cout << "Instance created" << endl;

    IMath* pMath = NULL;
    hr = pUnk->QueryInterface( IID_IMath, (LPVOID*)&pMath );
    pUnk->Release();
    if ( FAILED( hr ))
    {
       cout << "QueryInterface() for IMath failed" << endl;
       return -1;
    }

    long result;
    pMath->Multiply( 100, 8, &result );
    cout << "100 * 8 is " << result << endl;

    pMath->Subtract( 1000, 333, &result );
    cout << "1000 - 333 is " << result << endl;

    cout << "Releasing instance" << endl;
    pMath->Release();

    cout << "Shuting down COM" << endl;
    CoUninitialize();

    return 0;
}
```

В начале программы мы поместили файл заголовков IMATH.H из проекта сервера. Для определения GUID компонентов перед ним включен файл INITGUID.H. Функция main в первую очередь обеспечивает инициализацию библиотеки COM. В примере для определения CLSID используется ProgID компонента. Однако, прежде чем мы сможем вызвать CLSIDFromProgID, необходимо преобразовать ProgID (для которого используется кодировка ANSI) в строку с кодировкой Unicode. Все вызовы COM, OLE и ActiveX имеют встроенные реализации Unicode. Поэтому до передачи строк в любую API-функцию COM они должны быть преобразованы в вызовы с кодировкой Unicode.

После получения CLSID компонента вызываем функцию CoGetClassObject и запрашиваем указатель на интерфейс фабрики классов для компонента Math. После этого с помощью вызова CreateInstance создаем экземпляр компонента Math. Затем освобождаем интерфейс фабрики классов. Функция CreateInstance возвращает указатель на интерфейс IUnknown, с помощью которого мы в конце концов запрашиваем IMath. Получив указатель на него, используем сервисы компонента для выполнения некоторых простых операций.

По окончании всего этого мы освобождаем указатель на интерфейс IMath и вызываем CoUninitialize (это необходимо сделать до завершения работы приложения).

После ввода описанной выше программы включите в проект клиента файл CLIENT.CPP и постройте приложение. При пошаговой отладке клиента можно дойти даже до текста программы сервера. Не пожалейте времени и хорошо разберитесь в этих простых примерах клиента и сервера COM. Данные примеры помогут вам понять, что представляет собой COM-разработка. Теперь, после овладения основами COM, рассмотрим средство, несколько облегчающее COM-разработку, - библиотеку активных шаблонов.

Часть 5. Реализация проекта COM-сервера на ATL.

Библиотека активных шаблонов

Библиотека активных шаблонов (ATL) представляет собой основу для создания небольших COM-компонентов. В ATL использованы новые возможности шаблонов, добавленные в C++. Исходные тексты этой библиотеки поставляются в составе системы разработки Visual C++. Кроме того, в эту систему разработки введено множество мастеров Visual C++, что облегчает начальный этап создания ATL-проектов.

> **ПРИМЕЧАНИЕ**
> Версия 2.0 ATL работает с версией 4.2b Visual C++. Она не входит в основной пакет поставки названного продукта, но может быть получена на Web узле Microsoft по адресу: http://www.microsoft.com/visualc/prodinfo. ATL (Версия 2.1, которая в основных чертах представляет собой новый пакет Версии 2.0) включена как составляющая часть в пятую версию Visual C++.

Основные возможности библиотеки ATL

Библиотека ATL обеспечивает реализацию ключевых возможностей COM компонентов. Выполнения многих рутинных процедур, с которыми мы столкнулись при разработке последнего примера, можно избежать за счет использования классов шаблонов ATL. Приведем далеко не полный список функций ATL. Некоторые из них будут рассмотрены в этой главе.

- Утилита AppWizard, предназначенная для создания первичного ATL-проекта.
- Мастер объектов, используемый для добавления в проект компонентов различных типов.
- Поддержка по умолчанию основных интерфейсов COM, таких как IUnknown и IClassFactory.
- Поддержка механизма транспортировки пользовательского интерфейса.
- Поддержка базового механизма диспетчеризации (автоматизации) и двунаправленного интерфейса.
- Существенная поддержка разработки небольших элементов управления ActiveX.

Сравнение с библиотекой базовых классов фирмы Microsoft

Основной задачей ATL является облегчение создания небольших COM-компонентов. Задача MFC - ускорение разработки больших

Windows-приложений. Функции MFC и ATL несколько перекрываются, в первую очередь в области поддержки OLE и ActiveX.

Например, элементы управления ActiveX можно создавать как с помощью ATL, так и с использованием MFC. Применяя MFC и мастер элементов управления, можно создать полнофункциональный элемент управления, добавив всего несколько собственных строк программы к тем тысячам, которые предоставляет MFC. Однако при использовании этого элемента необходимо учитывать время его загрузки и выполнения. Дело в том, что главный исполняемый DLL-файл MFC имеет размер около одного мегабайта, что недопустимо в средах с узкой полосой пропускания, каковой является WWW. ATL также поддерживает элементы управления ActiveX. Однако в случае ее использования приходится самостоятельно писать большое количество программ, что нельзя сделать без знания COM и спецификации элементов управления ActiveX.

Это только один пример. В большинстве случаев при разработке COM-компонента с небольшим количеством или полным отсутствием визуальной информации вполне пригодна ATL. Если же приходится разрабатывать Windows-приложение с широкими функциями визуализации, то, вероятно, придется воспользоваться MFC. Мы высказали только общие соображения по этому вопросу и рекомендуем вам, прежде чем принимать окончательное решение, ознакомиться с обеими библиотеками.

Пример сервера

Чтобы проиллюстрировать некоторые свойства ATL, преобразуем пример сервера, написанный ранее на C++, так, чтобы в нем применились некоторые функции ATL. Для создания первоначального проекта воспользуемся утилитой ATL AppWizard пакета Visual C++.

Утилита ATL COM AppWizard

Утилита ATL COM AppWizard представляет собой мастер Visual C++, облегчающий поэтапное создание первоначального ATL-проекта (см. рис. 8 и табл. 10). К этой утилите при создании любого проекта приходится обращаться только один раз. Как только проект создан, для добавления в него компонентов применяйте мастер объектов ATL. Запустите Visual C++ и выполните следующие действия:

- Выберите пункт New... в меню File.
- Во вкладке Projects отметьте проект ATL COM AppWizard.
- Выберите подходящий каталог и назовите проект AutoSvr.
- Щелкните на кнопке Finish.

Файл	Описание
AUTOSVR.CPP	Главный файл проекта. В этом файле содержатся функции поддержки, необходимые COM для обеспечения хранения компонентов.
AUTOSVR.IDL	IDL-файл (файл описания интерфейсов) проекта. Сюда добавляются определения интерфейсов и методов. Этот файл будет обработан компилятором MIDL, после чего для проекта будет создана библиотека типов.
AUTOSVR.DEF	Файл определений Windows. Содержит описание точек входа DLL. Для EXE-проектов такой файл не создается.
AUTOSVR.RC	Файл ресурсов проекта.
STDAFX.H и STDAFX.CPP	Определения и файлы заголовков ATL.

Таблица 10. Файлы, создаваемые с помощью ATL AppWizard

Мастер объектов ATL

Созданный с помощью ATL AppWizard каркас проекта обеспечивает только хранение компонентов, но не содержит файлы, необходимые для создания того или иного компонента. Эти файлы можно создать, обратившись к мастеру объектов ATL, вызвать который можно, выбрав пункт **Add ATL Component...** меню **Insert** Visual C++.

Главное диалоговое окно мастера объектов показано на рис. 9. Существуют четыре различные категории объектов, которые могут быть включены в проект: ATL Control, ATL Miscellaneous и ATL Objects. К первой категории относятся элементы управления двух основных типов и объект Property Page. Ко второй категории относятся COM-объекты, представляющие собой диалоговые окна Windows. Для наших задач потребуется добавление простого объекта (Simple Object) из категории ATL Objects. После нажатия кнопки Next должно появиться диалоговое окно, изображенное на рис. 10.

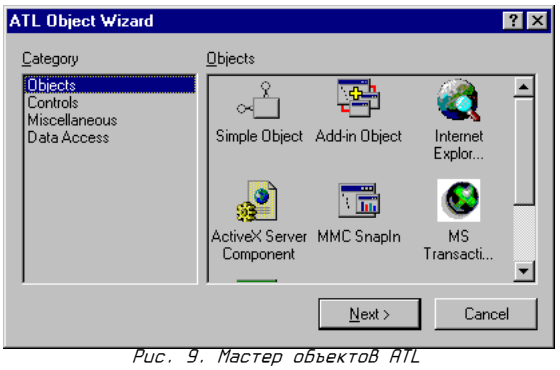

Рис. 9. Мастер объектов ATL

Имена, задаваемые в мастере объектов

В зависимости от типа выбранного объекта вам предлагается несколько диалоговых окон. При создании простого COM-объекта вся необходимая информация должна быть введена в двух диалоговых окнах **Name** и **Attribute**. Для более сложных объектов, таких как элементы управления ActiveX, требуется дополнительная информация. Более подробно речь об этом будет идти в следующей главе (при описании процедуры создания с помощью ATL элемента управления ActiveX).

На рис. 10 изображено диалоговое окно мастера объектов с активной вкладкой Names и с введенными значениями, соответствующими рассматриваемому объекту. В табл. 11 уточняется назначение каждого параметра. Для рассматриваемого примера зададим в качестве значения поля **Short Name** строку "Math". Кроме того, изменим предлагаемые названия разделов реестра **CoClass** и **ProgID**, присоединив к ним рабочее название "Component".

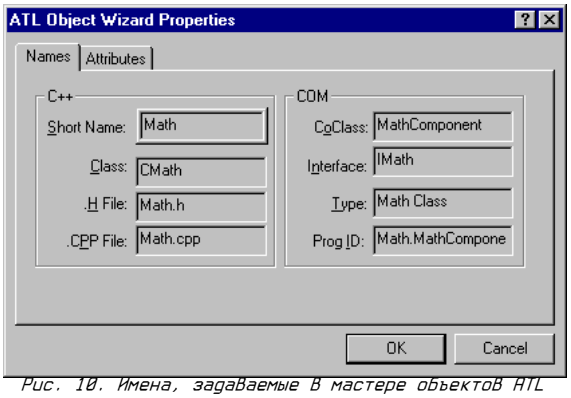

Рис. 10. Имена, задаваемые в мастере объектов ATL

Поле	Описание
Short Name	Данный элемент обеспечивает введение префикса для остальных элементов диалогового окна. Он непосредственно не связан ни с каким определенным атрибутом объекта. При изменении этого значения элементы оставшейся части страницы также изменятся.
Class	Имя C++-класса, реализующего объект.
Файлы H и CPP	Файлы заголовков и файлы реализации.
CoClass	Имя COM-класса, которое будет использовано внешними клиентами в качестве "типа" компонента.
Interface	Имя интерфейса, который требуется создать для объекта. Рассматриваемый нами объект будет предоставлять интерфейс IMath, который мы уже описали в этой главе.
Type	Удобное для восприятия имя компонента, размещаемое в реестре. При программировании значения не имеет.
ProgID	Программный идентификатор компонента. Клиенты могут его использовать для размещения и создания экземпляра компонента.

Таблица 6.11. Параметры диалогового окна Name мастера объектов

Атрибуты мастера объектов

Вкладка **Attributes** позволяет определить основные параметры хранилища компонента. Многие подробности, касающиеся почти каждого параметра, не являются предметом рассмотрения данной главы. Дело в том, что мы не рассматривали такие вещи, как потоковые COM-модели и понятие COM-агрегации, поэтому здесь значения этих параметров принимаются по умолчанию. На рис. 11 изображено диалоговое окно мастера объектов с активной вкладкой **Attributes**, а в табл. 12 подробно описаны все возможные параметры. После задания значений параметров щелкните на кнопке **OK**, в результате для нового объекта будут созданы исходные файлы.

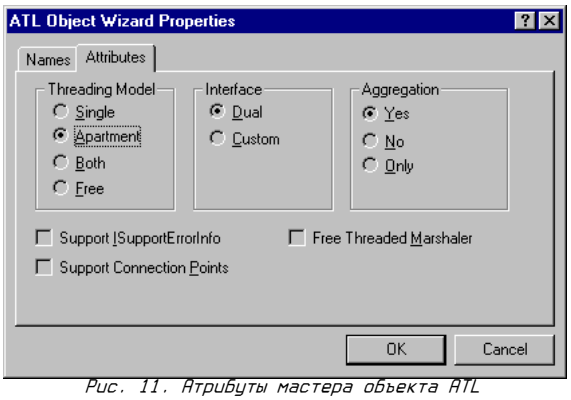

Рис. 11. Атрибуты мастера объекта ATL

Поле	Описание
Threading Model (потоковая модель)	COM-компоненты могут использовать следующие потоковые модели: *Простую* – компонент использует только один поток. *Модель изолированных потоков* – объекты расположены только внутри собственных потоков. *Двойную* – компонент может поддерживать как изолированные потоки, так и работу с произвольным количеством потоков. *Свободную* – компонент поддерживает работу с произвольным количеством потоков.
Interface (интерфейс)	Мы еще не рассматривали понятие двунаправленного интерфейса. Отметим, однако, что в соответствии с рекомендациями Microsoft компоненты по возможности должны его поддерживать. Двунаправленный интерфейс реализует как пользовательский, так и стандартный интерфейс автоматизации IDispatch. Это дает возможность клиенту выбирать способ доступа к функциям компонента.
Aggregation (агрегация)	Агрегация представляет собой широко используемый в COM технический прием, позволяющий включать и использовать в компоненте функции другого компонента. Внутренний компонент должен явно поддерживать эту технологию с помощью своей реализации интерфейса IUnknown. Значение данного атрибута позволяет компоненту выбирать, поддерживать агрегацию или нет.
Support ISupportErrorInfo (поддержка для интерфейса IsupportErrorInfo)	При установке данного атрибута мастер объекта добавит в проект реализацию интерфейса ISupportErrorInfo. Таким образом, будет обеспечен устойчивый механизм передачи сообщений об ошибках в системе клиент-сервер.

Support Connection Points (поддержка точек подключения)	При установке данного атрибута мастер объектов добавит в проект точку подключения интерфейсов COM. Эта процедура будет подробно рассмотрена позже.
Free-Threaded Marshaler (потоковый транспортировщик)	В проект добавляются средства транспортировки указателей интерфейсов через границы потоков.

Таблица 12. Атрибуты мастера объектов

Класс CComModule

Класс CComModule обеспечивает базовую поддержку создания хранилищ COM-объектов. Этот класс используется как в DLL-, так и EXE-реализациях. При генерации утилитой AppWizard главного файла проекта AUTOSVR.CPP в него добавляется глобальная переменная – экземпляр класса CComModule с именем _Module. Глобальные объекты создаются, как только модуль начинает выполняться.

В классе CComModule реализовано множество функций поддержки, большинство из которых обеспечивают базовую поддержку функций COM. Подробности можно найти в документации на ATL. Здесь мы рассмотрим только те методы, которые будут использованы в нашей реализации. Приведем файл AUTOSVR.CPP из рассматриваемого примера:

```
// AutoSvr.cpp : Implementation of DLL Exports.


// Note: Proxy/Stub Information
//      To build a separate proxy/stub DLL,
//      run nmake -f AutoSvrps.mk in the project directory.

#include "stdafx.h"
#include "resource.h"
#include "initguid.h"
#include "AutoSvr.h"

#include "AutoSvr_i.c"
#include "Math.h"

CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
   OBJECT_ENTRY(CLSID_MathComponent, CMath)
END_OBJECT_MAP()

////////////////////////////////////////////////////////////////////////////
// DLL Entry Point

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID /*lpReserved*/)
{
   if (dwReason == DLL_PROCESS_ATTACH)
   {
      _Module.Init(ObjectMap, hInstance);
      DisableThreadLibraryCalls(hInstance);
   }
   else if (dwReason == DLL_PROCESS_DETACH)
      _Module.Term();
   return TRUE;    // ok
}

////////////////////////////////////////////////////////////////////////////
// Used to determine whether the DLL can be unloaded by OLE

STDAPI DllCanUnloadNow(void)
{
   return (_Module.GetLockCount()==0) ? S_OK : S_FALSE;
}

////////////////////////////////////////////////////////////////////////////
// Returns a class factory to create an object of the requested type

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
   return _Module.GetClassObject(rclsid, riid, ppv);
}

////////////////////////////////////////////////////////////////////////////
// DllRegisterServer - Adds entries to the system registry

STDAPI DllRegisterServer(void)
{
   // registers object, typelib and all interfaces in typelib
   return _Module.RegisterServer(TRUE);
}

////////////////////////////////////////////////////////////////////////////
// DllUnregisterServer - Removes entries from the system registry

STDAPI DllUnregisterServer(void)
{

   return  _Module.UnregisterServer();
```

```
}
```

Прежде всего, в этом файле глобально объявлен класс CComModule. После этого следуют макросы ATL, в которых объявляются компонентные объекты, поддерживаемые в данном программном модуле. В нашем случае используется только один такой объект – компонент Math, программа для которого была создана ранее с помощью мастера объектов ATL. Макрос **BEGIN_OBJECT_MAP** определяет начало массива объявлений компонентных объектов. Для каждого компонента предусмотрена соответствующая строка макроопределения **OBJECT_ENTRY**. Макрос **OBJECT_ENTRY** содержит необходимую информацию, CLSID и имя внутреннего класса, необходимые для создания экземпляра компонента с помощью функции DllGetClassObject, предоставляемой для общего пользования.

В оставшейся нерассмотренной части файла реализованы четыре точки входа, необходимые COM для организации хранения компонентов в DLL-файле. Функция DllCanUnloadNow использует глобальный экземпляр CComModule для определения возможности выгрузки DLL-файла. Функции DllGetClassObject, DllRegisterServer и DllUnregisterServer работают так, как описано выше. Класс CComModule помогает нам управлять этими функциями.

При создании проекта мы использовали основные функции, предоставляемые утилитой AppWizard. Для добавления класса CMath был применен мастер объектов ATL. Таким образом были созданы файлы MATH.H и MATH.CPP. CPP-файл начинается следующими строками:

```
//
// Math.cpp : Implementation of CMath
//

#include "stdafx.h"
#include "AutoSvr.h"
#include "Math.h"

/////////////////
// CMath
/////////////////
```

Сюда осталось добавить только текст написанной нами программы. С файлом заголовков дело обстоит несколько сложнее. Мастер объектов ATL создал файл MATH.H в следующем виде:

```
////////////////////////////////////////////////////////////////////////////
// CMath
class ATL_NO_VTABLE CMath :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMath, &CLSID_MathComponent>,
   public IDispatchImpl<IMath, &IID_IMath, &LIBID_AUTOSVRLib>
{
public:
   CMath()
   {
   }

DECLARE_REGISTRY_RESOURCEID(IDR_MATH)

BEGIN_COM_MAP(CMath)
   COM_INTERFACE_ENTRY(IMath)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IMath
public:
};
```

В следующих разделах мы подробно рассмотрим использованные здесь макросы.

Макрос ATL_NO_VTABLE

Класс CMath унаследовал свойства многих классов шаблонов ATL. Прежде чем мы рассмотрим каждый из них, обратим внимание на некоторые макросы, использованные в объявлениях заголовков. Самый первый из них фактически эквивалентен директиве _declspec (novtable), которая сообщает компилятору о том, что для класса не требуется создавать виртуальную таблицу. Таким образом, намного уменьшается объем программы его конструирования или удаления.

При использовании макроса **ATL_NO_VTABLE** необходимо помнить о связанных с ним многочисленных ограничениях. В основном они касаются запрета на вызов чисто виртуальных функций или функций, которые приведут к вызову чисто виртуальной функции в конструкторе либо деструкторе класса. Если вы не уверены в соблюдении этих правил для вашего класса, то можете убрать из объявления макрос **ATL_NO_VTABLE**, поскольку он отвечает только за уменьшение размера генерируемого кода.

Макрос DECLARE_REGISTRY_RESOURCEID

Следующим макросом в программе является **DECLARE_REGISTRY_RESOURCEID**. Этот макрос преобразуется в вызов компонента ATL Registry, называемого также **Registrar**. Компонент Registrar предоставляет простой, управляемый данными механизм обновления содержимого регистра на основе определенной информации о компонентах. Для предоставления такой информации мастер объектов ATL создает файл *ComponentName.RGS* (в нашем случае это файл MATH.RGS), содержащий сценарий обновления реестра. Он имеет следующий вид:

```
HKCR
{
   Math.MathComponent.1 = s 'Math Class'
   {
      CLSID = s '{8C30BC11-B8F2-11D0-A756-B04A12000000}'
   }
   Math.MathComponent = s 'Math Class'
   {
```

```
      CurVer = s 'Math.MathComponent.1'
   }
   NoRemove CLSID
   {
      ForceRemove {8C30BC11-B8F2-11D0-A756-B04A12000000} = s 'Math Class'
      {
         ProgID = s 'Math.MathComponent.1'
         VersionIndependentProgID = s 'Math.MathComponent'
         ForceRemove 'Programmable'
         InprocServer32 = s '%MODULE%'
         {
            val ThreadingModel = s 'Apartment'
         }
      }
   }
}
```

Для описания разделов реестра каждого компонента в сценарии использован специальный синтаксис БНФ (нормальная форма Бекуса-Наура). Этот сценарий хранится в файле ресурсов проекта. Таким образом, макрос **DECLARE_REGISTRY_RESOURCEID** разворачивается в следующие строки программы:

```
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
   return _Module.UpdateRegistryFromResource(IDR_MATH, bRegister);
}
```

В конечном итоге приведенный выше метод вызывается обеими функциями, содержащимися в файле AUTOSVR.CPP: DllRegisterServer и DllUnregisterServer. Полное рассмотрение компонента Registrar выходит за рамки этой статьи. Однако следует отметить, что для изменения информации о каком-либо компоненте в реестре требуется отредактировать соответствующий RGS-файл и заново построить проект. Он будет рассматриваться как ресурс, и тогда при регистрации компонента информация реестра обновится.

Затем в нашем примере следуют такие макросы:

```
BEGIN_COM_MAP(CMath)
   COM_INTERFACE_ENTRY(IMath)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

> COM-объекты должны поддерживать саморегистрацию с помощью экспортируемых COM-функций DllRegisterServer и DllUnRegisterServer. В предыдущем примере обновление регистра происходило за счет REG-файла. При использовании сценария обновления регистра выполнять эту работу вручную не требуется, поскольку компонент ATL Registrar делает это за нас и, таким образом, для регистрации DLL-сервера можно воспользоваться следующей командной строкой: **regsvr32.exe server.dll**.

Макрос COM_INTERFACE_ENTRY

Как правило, каждый COM-объект поддерживает целый набор интерфейсов. Рассмотренный выше компонент Math поддерживает только два из них: IMath и IDispatch. Являющийся пользовательским для нашего компонента интерфейс IMath уже встречался выше, а IDispatch еще не рассматривался. С помощью этого интерфейса осуществляется поддержка для компонента механизма автоматизации, что делает использование данного компонента более удобным.

В любом случае макросы **BEGIN_COM_MAP** и **END_COM_MAP** создают таблицу интерфейсов, доступ к которой в компоненте осуществляется через интерфейс QueryInterface. Каждый интерфейс описывается с помощью макроса COM_INTERFACE_ENTRY. В следующей главе при рассмотрении вопросов создания элементов управления ActiveX будет описано большое количество интерфейсов. В простом примере, рассматриваемом сейчас, используются только два из них.

Таким образом, мы рассмотрели макросы, используемые в файле заголовков. Теперь обратимся собственно к определению класса:

```
class ATL_NO_VTABLE CMath :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMath, &CLSID_MathComponent>,
   public IDispatchImpl<IMath, &IID_IMath, &LIBID_AUTOSVRLib>
{
public:
   CMath()
   {
   } // IMath
public:
};
```

Класс CComObjectRootEx

Каждый класс, который является еще и COM-объектом, должен быть производным по отношению к одному из классов CComObjectRoot. Класс CComObjectRoot отвечает за подсчет обращений к компоненту. Из предыдущего фрагмента программы ясно, что поддержка интерфейса IUnknown обеспечивается путем наследования его от класса CComObjectRootEx.

> **ПРИМЕЧАНИЕ**
> Одним из наилучших источников информации об ATL является сама исходная программа. Такие библиотеки шаблонов, как ATL, поставляются с исходными текстами программ. Их можно найти в каталоге \DevStudio\VC\ ATL\Include.

Класс CComCoClass

Класс CMath также является производным от класса CComCoClass. Класс CComCoClass предоставляет рассматриваемому компоненту фабрику классов, а также основные методы, необходимые для получения CLSID и специфичной для компонента информации об ошибках. При разворачивании шаблона получим примерно следующее:

```
template <class T, const CLSID* pclsid = &CLSID_NULL>
class CComCoClass
{
public:
DECLARE_CLASSFACTORY()
DECLARE_AGGREGATABLE(T)
typedef T _CoClass
static const CLSID& WINAPI GetObjectCLSID() {return *pclsid;} static LPCTSTR WINAPI GetObjectDescription() {return NULL;} static
const IID& iid = GUID_NULL, HRESULT hRes = 0)
{
 return AtlReportError(GetObjectCLSID(), lpszDesc, iid, hRes);
}
...
};
```

Следует отметить, что класс CComCoClass предоставляет компоненту фабрику классов посредством макроса **DECLARE_CLASSFACTORY**. Этот макрос разворачивается, формируя класс, содержащий экземпляр одного из классов CComClassFactory. Остальные методы класса CComCoClass в основном отвечают за сообщения об ошибках.

Класс CComClassFactory

Большинство компонентов, использующих ATL, получают фабрику классов с помощью класса CComCoClass. Класс CComClassFactory предоставляет два стандартных метода фабрики классов: CreateInstance и LockServer. Несмотря на простоту реализации, класс CComClassFactory дает возможность не использовать всю программу фабрики классов, как это делалось в предыдущем примере. Взгляните на шаблон класса:

```
class CComClassFactory :
public IClassFactory,
public CComObjectRootEx<CComGlobalsThreadModel>
{
 public:
BEGIN_COM_MAP(CComClassFactory)
COM_INTERFACE_ENTRY(IClassFactory)
 END_COM_MAP()
// IClassFactory
STDMETHOD(CreateInstance)(LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj);
STDMETHOD(LockServer)(BOOL fLock);
…
};
```

Помните, фабрика классов сама по себе является COM-объектом, производным от класса CComObjectRootEx.

Класс IDispatchImp

Последним классом, который использовался в данном примере, является IDispatchImp. Его задачей было обеспечение для компонента реализации интерфейса IDispatch. Компонент содержит двунаправленный интерфейс, состоящий из пользовательского интерфейса (как в рассмотренном выше примере) и из полной реализации интерфейса OLE-автоматизации IDispatch. Для обеспечения работоспособности интерфейса IDispatch необходимо, чтобы в сервере были реализованы четыре метода: GetIDsOfNames, GetTypeInfCount, GetTypeInfo и Invoke. Однако благодаря ATL нам остается реализовать только собственные методы интерфейса IMath.

Интерфейс IDispatch

Обзор возможностей механизма автоматизации был дан в главе 5. Автоматизация представляет собой стандартный механизм, позволяющий осуществлять независимое от языка взаимодействие программных модулей. Чтобы обеспечить независимость от языка, большинство COM-компонентов поддерживают автоматизацию как механизм предоставления своих функциональных возможностей.

В основе автоматизации лежит COM-интерфейс IDispatch, обеспечивающий предоставление набора методов, позволяющих приложению клиента осуществлять динамический доступ к серверу автоматизации. Процедура динамического вызова процесса несколько отличается от COM-технологии вызова интерфейса виртуальных таблиц, использованного в примере компонента Math.

При использовании интерфейса виртуальных таблиц приложению клиента в ходе компиляции требуются сведения о состоянии интерфейса компонента, которые предоставляются либо через библиотеку типов, либо путем включения объявления интерфейса компонента в препроцессорную часть программы (например, в файл IMATH.H). Таким образом, реализуется раннее связывание интерфейса компонента.

Компонент, реализующий свои методы с помощью интерфейса IDispatch, отличается от пользовательского тем, что предоставляет много дополнительных возможностей. Взаимодействие между приложениями клиента и сервера может осуществляться с помощью *позднего связывания* методов интерфейса компонента. В этом случае клиенту не требуются объявления интерфейса сервера и т.п.

Таким образом, серверный компонент может легко изменить свой интерфейс (даже во время выполнения!) без необходимости перекомпиляции или перекомпоновки приложения клиента. Конечно, при изменении интерфейса сервера должен быть предусмотрен некоторый способ общения этого сервера с клиентом, чтобы последний смог воспользоваться новым интерфейсом.

Другой замечательной особенностью интерфейса IDispatch является стандартный способ транспортировки. Реализация COM по умолчанию, обеспечиваемая средой Windows, содержит набор типов данных, который может быть использован компонентами, применяющими интерфейс IDispatch. Внутренняя поддержка этих типов данных упрощает подключение как локальных, так и удаленных компонентов.

Большинство COM-интерфейсов похожи на построенный нами в примере интерфейс IMath. Они всегда определяются так, чтобы на

основе информации о них можно было точно описать соответствующий абстрактный класс. В COM абстрактный класс определен всегда (как интерфейс), а задачей разработчика является обеспечение его конкретной реализации. Интерфейс IDispatch имеет в этом смысле некоторые отличия, поскольку он придает рассматривавшимся до сих пор интерфейсам с виртуальными таблицами определенную "произвольность". Для описания интерфейса рассматриваемого типа используется модификатор dispinterface. Данный модификатор указывает на отличие такого интерфейса от стандартного (с виртуальными таблицами). Теперь клиент не получает доступ к функциональным возможностям компонента посредством указателя на виртуальную таблицу, как это было в примере с IMath, а вначале должен "подыскать" требуемую функцию, определить для нее некий внутренний идентификатор и, наконец, вызвать (выполнить) ее. Внимательно изучите рис. 12, чтобы уяснить суть этого процесса.

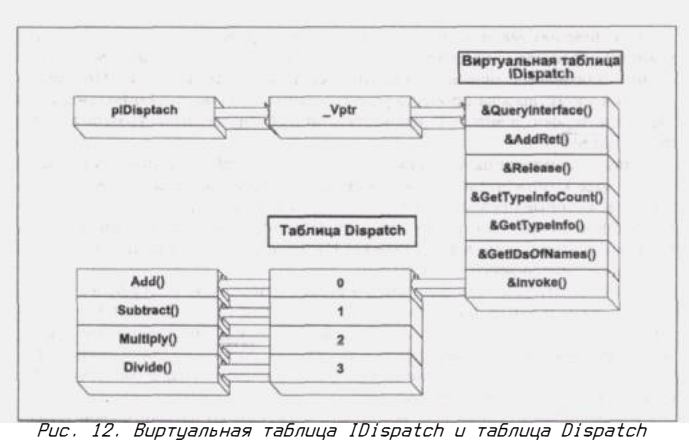

Рис. 12. Виртуальная таблица IDispatch и таблица Dispatch

Как видно из рисунка, интерфейс IDispatch выполняет достаточно много работы. Клиент все еще получает доступ к указателю на виртуальную таблицу, но теперь через нее уже нельзя получить прямой доступ к методам интерфейса IMath, а нужно сначала запросить функцию IDispatch::Invoke(). Связь вызова с определенным методом осуществляется через параметры метода Invoke(). Такой косвенный вызов обеспечивает механизм позднего связывания методов компонента. В таблице 13 описаны четыре метода интерфейса IDispatch.

Метод	Описание
Invoke()	Обеспечивает большую часть функциональных возможностей интерфейса IDispatch. Имеет восемь параметров, наиболее важным из которых является DISPID. DISPID связан с определенным смещением в таблице диспетчеризации и задает, какой метод компонентного объекта будет вызван.
GetIDsOfNames()	Обеспечивает контроллер возможностью связывать текстовое свойство сервера или имя метода, типа "Multiply", с численным параметром DISPID, который затем может быть использован в функции Invoke() для получения доступа к свойству или методу сервера.
GetTypeInfo()	Контроллер, обеспечивающий динамический поиск и вызов методов автоматизации, обычно не обладает всей информацией о типах, необходимой для формирования значений параметров метода Invoke. Серверу автоматизации необходимо вызвать метод GetTypeInfoCount() для определения возможности получения от компонента информации о типах и, если это, возможно, вызвать затем метод GetTypeInfo() для получения указанной информации.
GetTypeInfoCount()	Используется контроллером для выяснения, содержит ли компонентный объект необходимую контроллеру информацию о типах. Значение единица переданного параметра означает, что информация о типах доступна, а значение нуль – что недоступна.

Таблица 13. Методы интерфейса IDispatch

Двунаправленный интерфейс

Мы уже получили некоторые отрывочные знания, необходимые для понимания того, что такое двунаправленный интерфейс. Двунаправленный интерфейс реализуется серверным компонентом и предоставляет приложению клиента два различных способа доступа к своим функциональным возможностям. Он сочетает в себе интерфейс виртуальной таблицы (например, IMath) с интерфейсом IDispatch. Таким образом, клиент может выбирать, какой из интерфейсов ему, более удобен.

На рис. 13 показано, как будет выглядеть схема компонента Math с двунаправленным интерфейсом. Последний представляет собой сочетание пользовательского интерфейса IMath, реализованного через виртуальную таблицу, и интерфейса IDispatch, реализованного с помощью ATL. Методы компонента Math доступны непосредственно через виртуальную таблицу, *а также через* IDispatch.

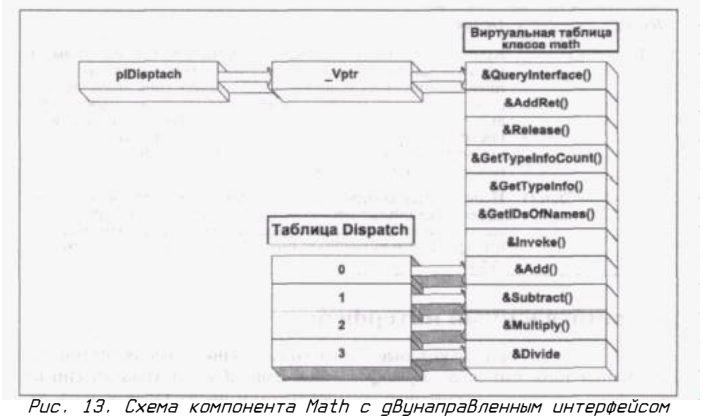

Рис. 13. Схема компонента Math с двунаправленным интерфейсом

Возникает вопрос: для чего нужно предоставлять два интерфейса, выполняющих практически одни и те же функции? Прежде всего, из соображений эффективности. Если сервер реализован как внутризадачный (DLL-файл), то не требуется никакой транспортировки.

Клиент может напрямую связаться с методами пользовательского интерфейса сервера и быстро произвести все необходимые вызовы. Эффективность этого метода сравнима с прямым связыванием функций C или C++.

Однако если клиенту требуется позднее связывание, то он может обратиться к реализации IDispatch. Такое обращение, конечно, происходит медленнее, поскольку большая часть работы производится во время выполнения, а не компиляции, но во многих случаях позднее связывание предпочтительнее. Ниже мы еще раз вернемся к технологиям связывания.

Добавление методов интерфейса

До сих пор рассматривались только функциональные возможности, обеспечиваемые средствами ATL. Сейчас мы опишем конкретные функциональные возможности создаваемого нами компонента. для этого с помощью мастера объектов в проект сервера будут добавлены два новых файла MATH.CPP и MATH.H, а также сделаны изменения в IDL-файле компонента, которые мы вскоре рассмотрим. Прежде всего, в новый класс следует добавить четыре метода.

```
class ATL_NO_VTABLE CMath :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMath, &CLSID_MathComponent>,
   public IDispatchImpl<IMath, &IID_IMath, &LIBID_AUTOSVRLib>
{
public:
   CMath()
   {
   }

DECLARE_REGISTRY_RESOURCEID(IDR_MATH)

BEGIN_COM_MAP(CMath)
   COM_INTERFACE_ENTRY(IMath)
   COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IMath
public:
   STDMETHOD(Add)( long, long, long* );
   STDMETHOD(Subtract)( long, long, long* );
   STDMETHOD(Multiply)( long, long, long* );
   STDMETHOD(Divide)( long, long, long* );
};
```

Все это вам должно быть уже известно. Сейчас мы просто добавили в объявление класса объявления простых математических функций Приведем реализацию этих функций в файле MATH.CPP, которая практически не отличается от предыдущей:

```
STDMETHODIMP CMath::Add( long op1, long op2, long* pResult )
{
   *pResult = op1 + op2;
   return S_OK;
}

STDMETHODIMP CMath::Subtract( long op1, long op2, long* pResult )
{
   *pResult = op1 - op2;
   return S_OK;
}

STDMETHODIMP CMath::Multiply( long op1, long op2, long* pResult )
{
   *pResult = op1 * op2;
   return S_OK;
}

STDMETHODIMP CMath::Divide( long op1, long op2, long* pResult )
{
   *pResult = op1 / op2;
   return S_OK;
}
```

Информация о типах

В рассмотренном ранее математическом примере любой клиент, желавший воспользоваться сервисами компонента, нуждался в некоторых предварительных сведениях об интерфейсе этого компонента. Информация о поддерживаемых интерфейсом возможностях, параметрах каждого метода и т.п. должна была быть получена из файла заголовков IMATH.H. Однако COM предоставляет и более универсальную технологию обеспечения клиентов подобной информацией. Она реализуется путем представления *информации о типах*.

Файл, содержащий информацию о типах компонента, создается с помощью языка определения интерфейса (Interface Definition Language - IDL). При создании хранилища компонента с помощью ATL создается также специальный IDL-файл, содержащий информацию о типах для каждого компонента, помещенного в хранилище. Содержимое этого файла преобразуется IDL-компилятором (MIDL.EXE) в двоичный формат и записывается в файл с расширением TLB, который называется библиотека типов. После компиляции информации о типах она должна быть помещена вместе с компонентом либо в самом хранилище, либо в виде отдельного TLB-файла. Способ размещения информации о типах компонента для приложений клиента или компонентных броузеров описывается в разделе **TypeLib** реестра.

Язык определения интерфейса

В основе языка определения интерфейса фирмы Microsoft лежит спецификация DCE RPC. В общем случае IDL используется для описания интерфейсов удаленного вызова процедур, но фирма расширила спецификацию, включив в этот язык поддержку для COM-интерфейсов. Одной из задач IDL, по крайней мере, в контексте COM-компонента, является определение интерфейса последнего

(т.е. его методов и параметров) независимо от языка программирования. Такое определение затем может быть использовано клиентами компонентов. Поскольку язык IDL позволяет описывать интерфейсы RPC, то он может предоставить для интерфейса компонента и соответствующую программу транспортировки, что дает возможность обходить межзадачные и сетевые ограничения.

> **ПРИМЕЧАНИЕ**
> До введения в обращение языка IDL Microsoft использовала язык описания объектов (Object Description Language – ODL), специально предназначенный для удовлетворения потребностей автоматизации. В настоящее время вместо него используется IDL, который более функционален и в то же время поддерживает устаревший ODL.

Синтаксис IDL аналогичен синтаксису языка C. Таким образом, определение интерфейса компонента очень напоминает объявление C ++-класса. В IDL не предусмотрены такие типы данных, как структуры, перечисления и т.д. Полное описание языка выходит за рамки этой главы, поэтому рассмотрим только основные его элементы, необходимые для построения примера. Ниже приводится программа файла AUTOSV.IDL. Добавьте к файлу выделенные строки.

```
//
// AutoSvr.idl : IDL source for AutoSvr.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (AutoSvr.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

   [
      object,
      uuid(8C30BC10-B8F2-11D0-A756-B04A12000000),
      dual,
      helpstring("IMath Interface"),
      pointer_default(unique)
   ]
   interface IMath : IDispatch
   {
   HRESULT Add( [in] long, [in] long, [out, retval] long* pResult );
   HRESULT Subtract( [in] long, [in] long, [out, retval] long* pResult );
   HRESULT Multiply( [in] long, [in] long, [out, retval] long* pResult );
   HRESULT Divide( [in] long, [in] long, [out, retval] long* pResult );
   };
[
   uuid(8C30BC01-B8F2-11D0-A756-B04A12000000),
   version(1.0),
   helpstring("AutoSvr 1.0 Type Library")
]
library AUTOSVRLib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   [
      uuid(8C30BC11-B8F2-11D0-A756-B04A12000000),
      helpstring("Math Class")
   ]
   coclass MathComponent
   {
      [default] interface IMath;
   };
};
```

В первой части программы определяется двунаправленный интерфейс IMath. IDL-определениям предшествует блок атрибутов, заключенный в квадратные скобки ([ ]), Данные атрибуты обеспечивают добавочную информацию к последующим определениям. Определение интерфейса IMath начинается несколькими строками атрибутов:

```
[
   object,
   uuid(8C30BC10-B8F2-11D0-A756-B04A12000000),
   dual,
   helpstring("IMath Interface"),
   pointer_default(unique)
]
```

Атрибут объекта задает COM-, а не RPC-интерфейс. Затем ключевое слово uuid задает GUID интерфейса. Ключевое слово dual обозначает, что интерфейс предоставляет как пользовательский интерфейс с виртуальной таблицей, так и стандартный интерфейс автоматизации IDispatch. Ключевое слово helpstring определяет текст, который может быть отображен броузером объекта. Атрибут POINTER_DEFAULT устанавливает атрибут по умолчанию для всех указателей, определенных в интерфейсе. Далее следует определение интерфейса.

```
interface IMath : IDispatch
{
HRESULT Add( [in] long, [in] long, [out, retval] long* pResult );
HRESULT Subtract( [in] long, [in] long, [out, retval] long* pResult );
HRESULT Multiply( [in] long, [in] long, [out, retval] long* pResult );
HRESULT Divide( [in] long, [in] long, [out, retval] long* pResult );
};
```

Данный фрагмент описывает COM-интерфейс компонента. Это практически полностью соответствует объявлению C++-класса, использованному ранее. Основное отличие состоит в перечислении параметров. В IDL существует несколько ключевых слов, применимых к параметрам метода. Ключевые слова in и out определяют *направленность* параметров (т.е. Входные они или Выходные). Таким образом, COM снабжается информацией, необходимой для повышения эффективности процесса транспортировки параметров. Ключевое слово retval определяет тот факт, что параметр должен трактоваться как тип значения, возвращаемого методом. Это играет роль при определении интерфейса для нашего элемента. Остальные строки IDL файла касаются хранилища и сохраняемых в нем объектов.

```
library AUTOSVRLib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   [
      uuid(8C30BC11-B8F2-11D0-A756-B04A12000000),
      helpstring("Math Class")
   ]
   coclass MathComponent
   {
      [default] interface IMath;
   };
};
```

Последний блок атрибутов описывает библиотеку типов в целом. В нем содержится информация о GUID, версии и строке помощи, которая используется приложениями поиска компонентов. Ключевое слово library задает имя библиотеки, а последующий блок, как правило, включает все определения, касающиеся конкретного хранилища. В этот блок входит информация о модуле интерфейса, типе и определении компонента. В рассматриваемом примере сервер содержит только компонент Math, заданный ключевым словом coclass. Функция ключевого слова coclass состоит в определении, отдельных компонентов и поддерживаемых ими интерфейсов. Компонент Math предоставляет двунаправленный интерфейс IMath, который принимается по умолчанию. В табл. 14 приводятся основные ключевые слова IDL и их краткое описание.

Ключевое слово	Описание
Object	Является начальным ключевым словом в определении пользовательского интерфейса. За ним следуют несколько атрибутов, описывающих дополнительные возможности интерфейса.
Uuid	GUID, однозначно определяющий заданный интерфейс, библиотеку типов или компонентный объект.
Dual	Указывает на двунаправленность интерфейса. Двунаправленный интерфейс предоставляет как интерфейс с виртуальной таблицей, так и IDispatch-интерфейс. Все методы двунаправленного интерфейса должны возвращать HRESULT, но могут для определения возвращаемого значения использовать также ключевое слово retval.
Helpstring	Задает строку, которая может отображаться на экране средствами просмотра компонента и интерфейса.
Interface	Задает фактическое имя интерфейса, которое затем используется в блоке coclass для определения поддерживаемых компонентом интерфейсов.
Coclass	Описывает интерфейсы, поддерживаемые в данном COM-объекте. GUID идентифицирует компонент в целом.
Default	Задает интерфейс компонента по умолчанию. Компонентный объект может иметь максимум два интерфейса по умолчанию – один в качестве исходного передающего интерфейса, а второй – в качестве принимающего.
in/out/retval	В вызовах методов это ключевое слово определяет направленность каждого параметра (т.е. Входной он или Выходной). Ключевое слово retval описывает параметр, который должен трактоваться как возвращаемое значение метода.

Таблица 14. Основные ключевые слова IDL

Создание сервера

Теперь мы полностью подготовлены к созданию проекта сервера. Благодаря ATL нам остается добавить лишь небольшое количество строк программы по сравнению с примером, написанным на C++. ATL предоставляет основное место хранения, счетчик обращений, фабрику классов и поддержку саморегистрации. Все это в предыдущем примере нам приходилось реализовывать самостоятельно.

Для быстрого тестирования компонента AutoSvr можно запустить утилиту OLEVIEW, установить для нее режим **Expert Mode** и открыть раздел **Automation Objects**. Компонент получит строковое имя **Math Class**. Откройте в дереве узел класса Math, и утилита OLEVIEW попытается загрузить компонент. Если все пойдет нормально, то на экране появятся имена трех поддерживаемых компонентом интерфейсов (рис. 14).

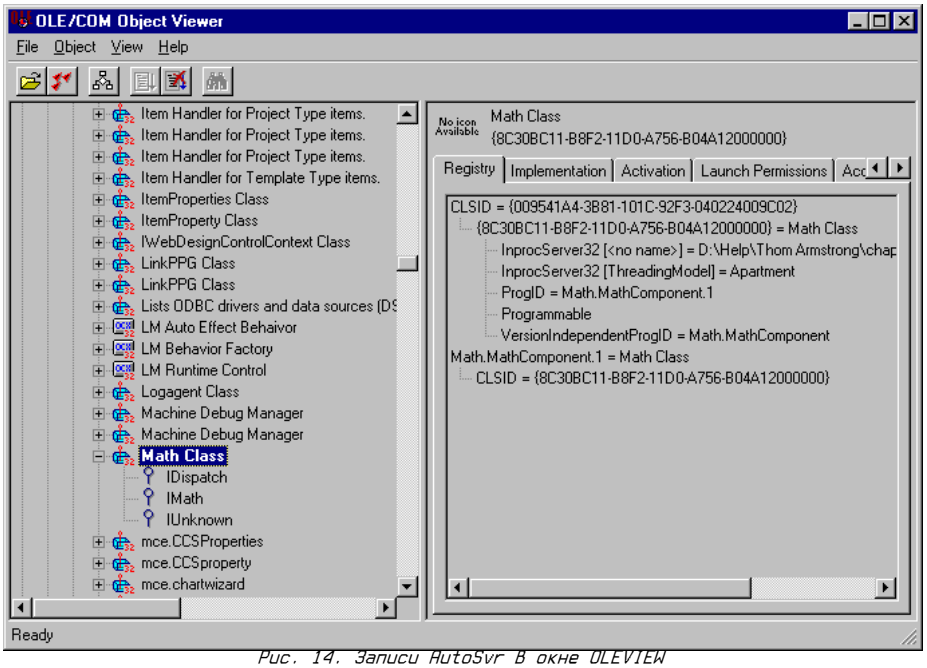

Рис. 14. Записи AutoSvr в окне OLEVIEW

Клиентская программа на Visual Basic

Для проведения тестирования разработанного нами ATL сервера создадим с помощью Visual Basic приложение клиента. Visual Basic является мощным средством разработки COM-приложений, в особенности тех из них, в которых использована автоматизация. Автоматизация – это как раз та технология, которая наиболее сочетаема с Visual Basic. Каждый объект Visual Basic, будь то элемент управления ActiveX, форма или класс Visual Basic, представляет собой компонент автоматизации. Поскольку среда Visual Basic приобретает все большую популярность, то поддержка разрабатываемыми вами компонентами механизма автоматизации имеет огромное значение.

Visual Basic предоставляет не только удобную среду поддержки автоматизации, но также и поддержку COM-интерфейсов с виртуальными таблицами. Мы уже рассматривали различия между интерфейсом с виртуальной таблицей и интерфейсом, в основе которого лежит стандартный интерфейс IDispatch. Теперь эти различия проявятся в действии.

Запустите Visual Basic и проделайте следующее:

1. Создайте новый проект типа Standard EXE.
2. Создайте форму с тремя полями ввода, имена которых будут **txtOp1**, **txtOp2** и **txtOp3**.
3. Добавьте три кнопки с именами **cmdDynamic**, **cmdIDBinding** и **cmdStatic**.
4. Слева от каждой кнопки разместите еще три поля ввода. Используйте массив элементов управления с именем txtTime.
5. По окончании этих манипуляций форма приобретет вид, представленный на рис. 15.

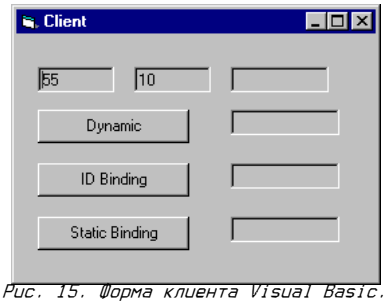

Рис. 15. Форма клиента Visual Basic.

Задачей нашего примера является демонстрация трех типов связывания интерфейсов автоматизации, поддерживаемых Visual Basic: позднее, ID-связывание и раннее. В документации по Visial Basic эти технологии называются соответственно поздним, ранним и очень ранним связыванием. При получении клиентским приложением доступа к функциональным возможностям компонента оно использует одну из этих технологий. Для применения первых двух технологий от компонента требуется предоставление интерфейса IDispatch, а для применения третьей необходимо, чтобы компонент реализовывал двунаправленный интерфейс. В рассмотренном компоненте Math все эти условия соблюдаются. Следовательно, имеется возможность воспользоваться любой из этих технологий.

Позднее (или динамическое) связывание

Позднее связывание является наиболее мощным средством автоматизации, позволяющее приложению клиента определять функциональные возможности компонента во время выполнения. В период разработки клиента имена методов и типы параметров не используются. Соответствующие функциональные возможности запрашиваются во время выполнения.

Другими словами, приложение клиента будет запрашивать идентификатор DISPID необходимого метода сервера для последующего вызова метода Invoke () с помощью интерфейса IDispatch. Все это происходит во время выполнения клиента. Этот способ требует больших затрат системных ресурсов и не обеспечивает проверку типов во время компиляции. Такая проверка осуществляется сервером уже во время выполнения, *при вызове метода*. Если же переданы данные неправильного типа, то выдается сообщение об ошибке.

Поскольку эта технология предусматривает прохождение всех процессов во время выполнения, то она является наиболее медленной из трех, однако в то же время и наиболее гибкой. Интерфейс сервера может измениться, при этом клиент не потребует перекомпиляции для получения новых возможностей, предоставляемых сервером после внесенных изменений. Для иллюстрации позднего связывания добавим в проект Visual Basic следующую программу вызов её происходит нажатием кнопки **Dynamic**.

```
Const ITERATIONS = 40000

Private Sub cmdDynamic_Click()
    ' Пример позднего связывания
    Dim obj As Object
    Dim Start, i  As Long
    Set obj = CreateObject("Math.MathComponent.1")
    Start = Timer
    txtResult = obj.Add(txtOp1, txtOp2)
    For i = 1 To ITERATIONS
        obj.Add txtOp1, txtOp2
    Next
    txtTime(0) = Timer - Start
    Set obj = Nothing
End Sub
```

Прежде всего, мы здесь объявляем переменную типа Object. В Visual Basic, таким образом, подключается компонент, поддерживающий интерфейс IDispatch. После этого с помощью вызова CreateObject получаем указатель на IDispatch и предаем идентификатор ProgID компонента. Константа **ITERATIONS** объявлена в части программы, находящейся вне процедур (блок GENERAL). Цикл For использован для многократного вызова метода Add () компонента. В программу включен таймер, который отображает на экране время, затраченное на каждую итерацию. По окончании каждой итерации устанавливаем значение переменной object равным nothing и таким образом освобождаем интерфейс IDispatch компонента.

Это - наиболее "дорогостоящая" технология реализации возможностей автоматизации компонента. Однако она позволяет достигнуть большой гибкости, поскольку вызов метода Add() в Visual Basic осуществляется динамически.

ID-связывание (или раннее связывание)

В предыдущем примере для каждого вызова метода Add () компонента Visual Basic приходилось вызывать метод IDispatch: :GetIDsOfNames (). При этом не могла быть проведена проверка типов, поскольку Visual Basic не обладал никакой информацией о типах компонента. Вторая технология связывания, называемая ID-связыванием, обеспечивает высокую эффективность и безопасность проверки типов во время компиляции. Чтобы применить эту технологию Visual Basic, потребуется загрузить библиотеку типов компонента.

Добавление в проект библиотеки типов осуществляется с помощью команды References Visual Basic. После этого ссылку на объект можно трактовать как тип данных Visual Basic. При компиляции проекта Visual Basic проверит синтаксис и сверит параметры, руководствуясь информацией о типах компонента, а также сохранит идентификатор DISPID для каждого метода и свойства. Таким образом, устраняется необходимость запроса DISPID для каждого метода во время выполнения. Одним из недостатков этого подхода является необходимость перекомпиляции при изменении интерфейса компонента. Приведем программу, реализующую пример ID-связывания.

```
Private Sub cmdIDBinding_Click()
    ' Пример ID-связывания (раннего связывания)
    Dim obj As MFCMathComponent
    Dim Start, i  As Long
    Set obj = New MFCMathComponent
    Start = Timer
    txtResult = obj.Add(txtOp1, txtOp2)
    For i = 1 To ITERATIONS
        obj.Add txtOp1, txtOp2
    Next
    txtTime(1) = Timer - Start
End Sub
```

Данная программа ссылается на новый класс MFCMathComponent. Если в компоненте доступен пользовательский интерфейс IMath, то Visual Basic будет использовать именно его. Чтобы оценить время, затрачиваемое на ID-связывание, создается MFC-сервер автоматизации, имитирующий работу компонента AutoSvr. MFC-компоненты автоматизации в общем случае не поддерживают двунаправленный интерфейс. Таким образом, связываясь с библиотекой типов во время компиляции, Visual Basic обеспечивает наибольшую эффективность программы. Это достигается кэшированием DISPID для каждого метода.

Раннее связывание (или очень раннее связывание)

Для раннего связывания необходимо, чтобы сервер предоставлял информацию о типах. Такая технология является наиболее эффективной, но и наименее гибкой. Раннее связывание обеспечивает проверку типов, чтобы уже на этапе компиляции клиент смог использовать информацию о типах для проверки параметров и возвращаемых значений. Связывание методов осуществляется посредством виртуальной таблицы и, таким образом, не требуется никаких идентификаторов DISPID либо вызовов функций GetIDsOfNames() или Invoke(). Если сервер реализован в виде DLL, как в нашем случае, то скорость выполнения раннего связывания сравнима со скоростью прямого вызова DLL-функции. Приведем программу очень раннего списывания для рассматриваемого интерфейса IMath.

```
Private Sub cmdStatic_Click()
    ' Пример очень раннего связывания
    Dim obj As New MFCMathComponent
    Dim Start, i  As Long
    Start = Timer
    txtResult = obj.Add(txtOp1, txtOp2)
    For i = 1 To ITERATIONS
        obj.Add txtOp1, txtOp2
    Next
    txtTime(2) = Timer - Start
End Sub
```

После ввода всех этих программ вы можете провести собственную проверку интерфейсов на эффективность. В табл. 15 приведено сравнение полученных нами результатов.

Метод связывания	Время (40000 итераций)
Динамическое	2424219
ID(раннее)	1740234
Посредством виртуальной таблицы (очень раннее)	1088672

Таблица 6.15. Сравнение эффективности интерфейсов

Заключение

Модель компонентного объекта (COM) фирмы Microsoft является основой таких технологий, как OLE и ActiveX. Модель COM принесла в разработку программ некоторые вещи, необходимость в которых ощущалась на протяжении длительного времени. COM наряду с независимостью от расположения предоставила для программных компонентов еще и независимый от языка программирования двоичный стандарт. Под независимостью от расположения понимается возможность размещения программных модулей независимо от решаемых задач и используемых для этого компьютеров. OLE и ActiveX представляют собой надстроенные над COM технологии прикладного уровня.

COM-объекты аналогичны объектам C++ в смысле инкапсуляции функциональных возможностей посредством предоставления хорошо отработанного интерфейса. В то же время они отличаются от C++ -экземпляров в одном важном аспекте: они предоставляют, как правило, более одного интерфейса. COM-объекты могут быть реализованы практически на любом языке программирования, но наиболее удобным для этой цели является C++, поскольку в COM используется понятие виртуальной таблицы функций. Каждый интерфейс COM должен реализовывать интерфейс IUnknown, который снабжает любой интерфейс компонента средствами добавления новых интерфейсов и счетчиком обращений, управляющим временем жизни компонента. COM-объекты функционируют внутри других компонентов-контейнеров, которыми могут быть другие приложения. В Windows-системах COM-объекты хранятся в DLL- или исполняемых файлах. Место их размещения получило название *хранилище компонента*. Каждое хранилище должно реализовывать стандартную технологию COM по предоставлению фабрики классов, которая, в свою очередь, является COM-компонентом, предназначенным для создания экземпляров других COM-объектов.

В среде COM большое значение имеет реестр Windows. Его роль состоит в управлении несколькими разделами, используемыми как COM, так и программами клиента. В COM широко используется понятие GUID - уникального 128-разрядного числа, однозначно идентифицирующего интерфейсы и компоненты. В состав COM входят несколько Win32 API и большой набор объявлений интерфейсов.

Активная библиотека шаблонов (ATL) представляет собой структуру, облегчающую создание небольших COM компонентов. В ATL использованы новейшие средства создания шаблонов C++. Она снабжена также исходной программой, представляющей собой часть среды разработки Visual C++. Создание проекта начинается с определения способа реализации хранилища. В этом существенную помощь оказывает утилита ATL COM AppWizard. После этого для добавления определенных компонентов используется мастер объектов ATL. В состав ATL входят несколько классов, обеспечивающих реализацию по умолчанию наиболее общих потребностей COM. Класс CComModule обеспечивает основную поддержку по организации хранилища посредством реализации функций DllGetClassObject и DllCanUnloadNow. Благодаря компоненту ATL Registrar обеспечивается также поддержка саморегистрации, что облегчает обновление регистра Windows.

Автоматизация представляет собой стандартный механизм, позволяющий осуществлять независимое от языка взаимодействие программных модулей. Чтобы удовлетворять этому важному требованию, большая часть COM-компонентов поддерживает автоматизацию как метод предоставления функциональных возможностей. Автоматизация поддерживается посредством COM-интерфейса IDispatch, который предоставляет целый набор методов, позволяющих приложению клиента динамически осуществлять доступ к функциональным возможностям сервера. Двунаправленный интерфейс сочетает в себе пользовательский COM-интерфейс с интерфейсом IDispatch. Описание интерфейсов COM-объектов происходит с помощью задания информации о типах. Файл, содержащий эту информацию, создается с использованием языка определения интерфейса (IDL) Microsoft и компилируется MIDL-компилятором. При этом создается двоичная библиотека типов, которая затем передается вместе с компонентом.

Примеры к главе 6

- SERVER - COM-сервер, написанный на чистом C++
- CLIENT - COM-клиент, написанный на чистом C++ - консольное приложение Win32.
- AUTOSVR - Сервер автоматизации, написанный на ATL. Предоставляет дуальный интерфейс IMath.
- MFCServer - Сервер автоматизации на MFC. Демонстрирует ID-связывание.
- VBClient - Клиент автоматизации, написанный на Visual Basic. Демонстрирует различные виды связывания.