Системное программирование

Лекция 5

Component Object Model

План лекции

- Что такое Component Object Model?
- Структура COM-приложений
- Интерфейсы IUnknown и IClassFactory
- Порядок разработки COM-приложения

Component Object Model

При разработке повторно используемого программного обеспечения, системный программист берет уже существующую или предлагает новую систему соглашений, которой оно должно соответствовать. Соглашения могут быть оформлены в виде спецификаций или корпоративных стандартов. В этих документах, как правило, оговариваются принципы именования объектов (имена функций, параметров, переменных), структуры и типы используемых данных, система интерфейсов (группы функций, классифицированных по каким-то признакам) и т. д.

Примером такой спецификации может служить **COM** (**Component Object Model** – объектная модель компоненты) компании Microsoft

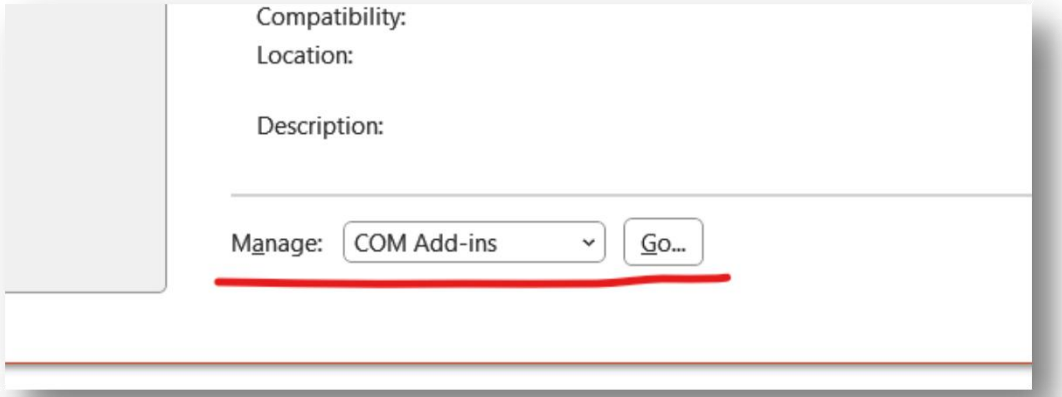
Component Object Model

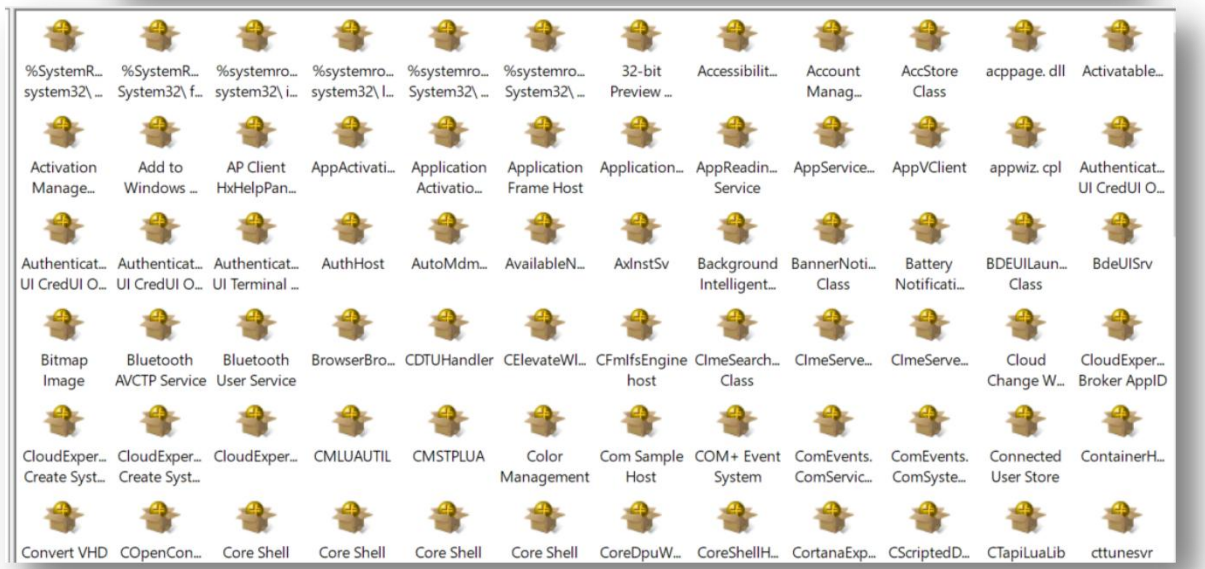
Component Object Model – объектная модель компоненты фирмы Microsoft является, как следует из её названия, ***моделью*** для проектирования и создания компонентных объектов. Модель определяет множество технических приемов, которые могут быть использованы разработчиком при создании независимых от языка 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

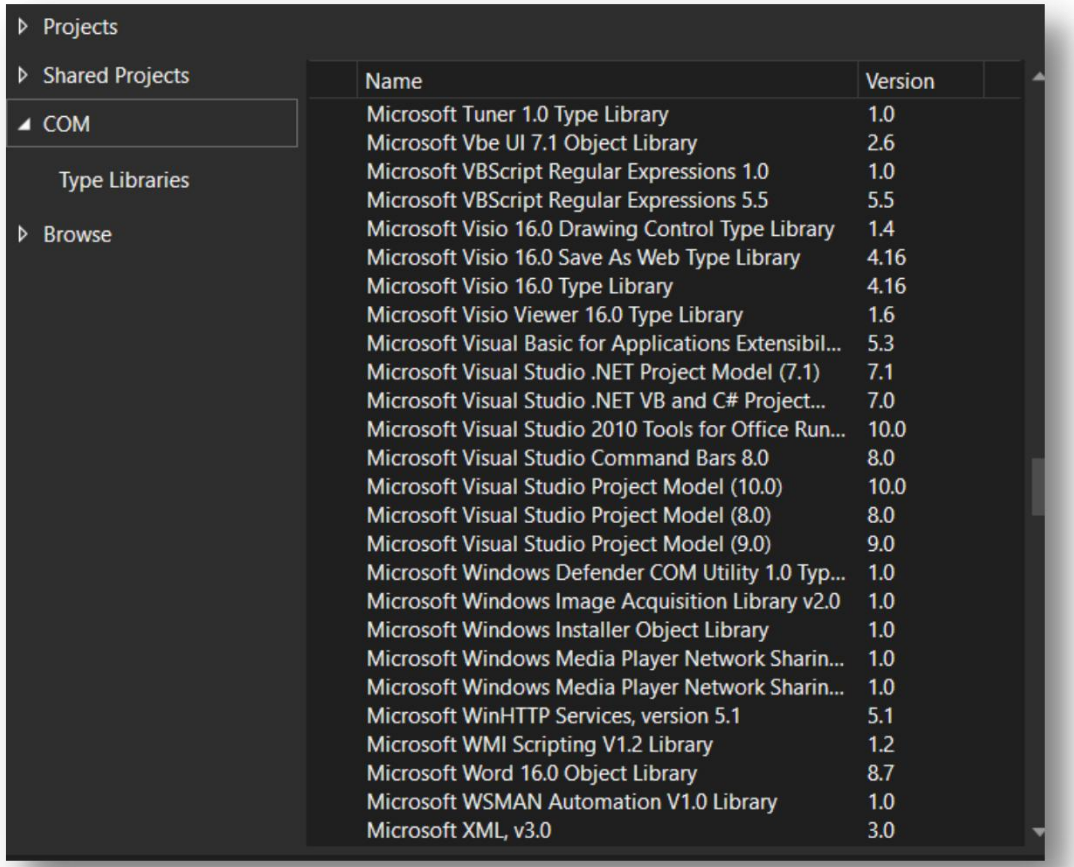
Данная модель была разработана как развитие техник разработки модульных приложений на языке C++

Component Object Model

Component Object Model в Windows повсюду







Component Object Model

Двоичный стандарт (или независимость от ЯП)

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

Component Object Model

Двоичный стандарт (или независимость от ЯП)

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++

Component Object Model

Двоичный стандарт (или независимость от ЯП)

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

Component Object Model

Двоичный стандарт (или независимость от ЯП)

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

Component Object Model

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

Component Object Model

Независимость от местоположения

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)

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

Component Object Model

Согласно самой компании Microsoft**, COM** – это независимая от платформы, распределенная, объектно-ориентированная система для создания бинарных программных компонентов, которые могут взаимодействовать между собой

Чтобы понять COM (и, следовательно, все технологии, основанные на COM), важно понимать, что это не объектно-ориентированный язык, а **стандарт**

В COM также не указано, как должно быть структурировано приложение: язык, структура и детали реализации остаются на усмотрение разработчика приложения. Скорее, COM определяет объектную модель и требования к программированию, которые позволяют COM-объектам взаимодействовать с другими объектами

Component Object Model

Как уже говорилось компоненты могут быть написаны на разных языках и могут быть совершенно разными по структуре, поэтому COM называют **двоичным стандартом –** стандартом, который применяется после того, как программа переведена в двоичный машинный код

Единственным языковым требованием COM является то, что код генерируется на языке, который может создавать структуры указателей и, явно или неявно, вызывать функции с помощью указателей

COM-программирование – разработка программного обеспечения, имеющего модель согласно спецификации COM

Component Object Model

Соответственно, первым основным понятием, которым оперирует стандарт COM, является **COM-компонент (COM-объект)**, представляющий собой программный модуль

COM-объект можно сравнить с объектом в понимании C++ или Java. Объект COM – это некоторая сущность, имеющая состояние и методы доступа, позволяющие изменять это состояние

COM-объекты можно создавать прямым вызовом специальных функций, но напрямую уничтожить его невозможно. Вместо прямого уничтожения используется механизм самоуничтожения, основанный на подсчете ссылок

Component Object Model

Так, в COM присутствует понятие класса. Класс в COM носит название **CoClass**

CoClass – это класс, поддерживающий набор методов и свойств (один или более), с помощью которых можно взаимодействовать с объектами этого класса. Такой набор методов и свойств называется **COM-интерфейсом** (Interface)

Каждый CoClass имеет два идентификатора – один из них, текстовый, называется **ProgID** и предназначен для человека, а второй, бинарный, называется **CLSID**

Component Object Model

CLSID является глобально уникальным идентификатором (**GUID**). GUID имеет размер 128 бит и уникален в пространстве и времени. Его уникальность достигается путем внедрения в него информации об уникальных частях компьютера, на котором он был создан, таких, как номер сетевой карты, и времени создания с точностью до миллисекунд

С помощью CLSID можно точно указать, какой именно объект требуется. Тип данных GUID применяется и для идентификации COM-интерфейсов. В этом случае он называется **IID**

```
// {D3F297B4-0C5F-4F8C-B4B8-D18C6DDD62C4}
static const GUID CLSID =
{ 0xd3f297b4, 0xc5f, 0x4f8c, { 0xb4, 0xb8, 0xd1, 0x8c, 0x6d, 0xdd, 0x62, 0xc4 } };
```

Component Object Model

В понимании COM интерфейс – это контракт, состоящий из списка связанных прототипов функций, чье назначение определено, а реализация – нет!

Эти прототипы функций эквивалентны абстрактным базовым классам C++, то есть классам, имеющим только **виртуальные методы**, описания без реализации

Определение интерфейса описывает функции-члены интерфейса, называемые методами, типы их возвращаемого значения, число и типы их параметров, а также описывает, что они, собственно, должны делать

Напрямую с интерфейсом не ассоциировано никакой реализации!!!

Component Object Model

Реализация интерфейса (interface implementation) – это код, который программист создает для выполнения действий, оговоренных в определении интерфейса

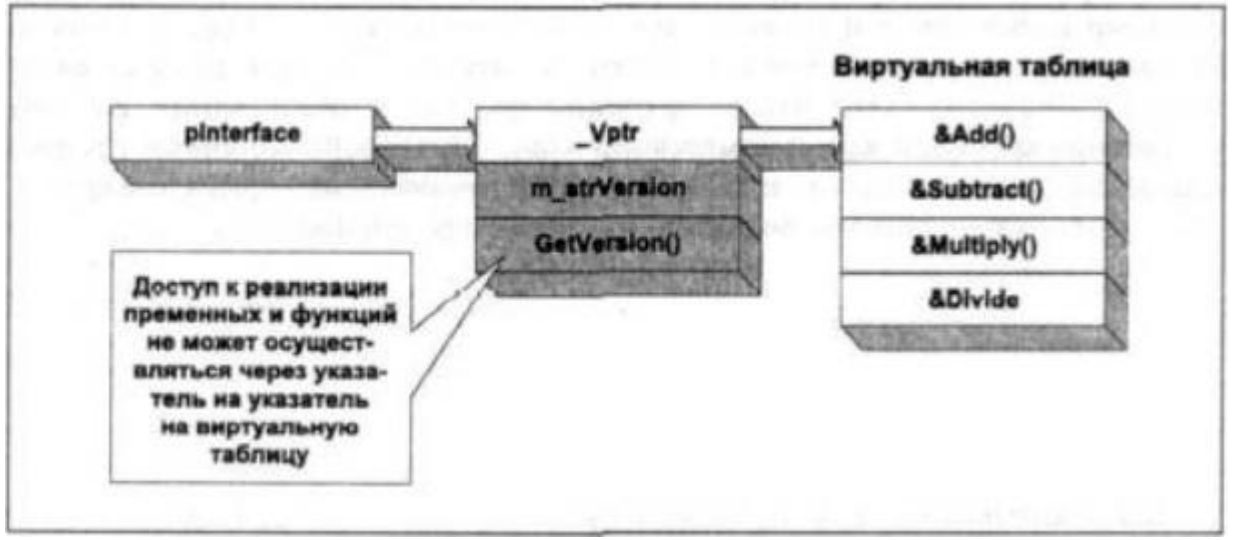
Реализации интерфейсов, помещенные в COM-библиотеки или EXE-модули, могут использоваться при создании объектно-ориентированных приложений. Разумеется, программист может игнорировать эти реализации и создать собственные

Интерфейсы ассоциируются с CoClass'ами. Чтобы воспользоваться реализацией функциональности интерфейса, нужно создать экземпляр объекта соответствующего класса, и запросить у этого объекта ссылку на соответствующий интерфейс

Component Object Model

Экземпляр «реализации интерфейса» на самом деле является указателем на массив указателей на методы (таблицу функций, ссылающуюся на реализации всех методов, определенных в интерфейсе, также называемую **виртуальной таблицей**)

Объекты с несколькими интерфейсами могут предоставлять указатели на несколько таблиц функций. Любой код, содержащий указатель, через который он имеет доступ к массиву, может вызывать методы этого интерфейса



Component Object Model

Слово «интерфейс» используется в COM не в том смысле, что в C++. Интерфейс в C++ ссылается на все функции, поддерживаемые классом. COM-интерфейс же ссылается на предварительно оговоренную группу связанных функций, реализуемых COM-классом, но **не обязательно на ВСЕ функции**, поддерживаемые классом

Интерфейсы бывают двух типов: **стандартные** и **произвольные**

За стандартными интерфейсами закреплены предопределенные GUID-идентификаторы. Важнейшим среди стандартных интерфейсов является интерфейс **IUnknown**

Все остальные интерфейсы являются производными (наследуют все методы) от **IUnknown**. Каждый компонент должен поддерживать (часто говорят «реализовывать») как минимум стандартный интерфейс **IUnknown**

Component Object Model

Как можно заметить по названию стандартного интерфейса – в COM стало доброй традицией начинать названия интерфейсов с «I»

Среди стандартных интерфейсов также стоит выделить **IClassFactory**

Данный интерфейс отвечает за управление жизненным циклом компонентов путём реализации паттерна «фабрика классов»

Данный интерфейс не является обязательным к реализации разрабатываемыми компонентами, но с ним становится проще следить за тем какие объекты были создзданы и когда можно их выгружать из памяти

Component Object Model

Рассмотрим создание однокомпонентного COM-сервера:

1. Описать COM-интерфейс который обязан наследоваться хотя бы от **IUnknown.** Тут есть два пути - с помощью языка IDL и скомпилировать его используя MIDL или на языке C/C++ напрямую

В итоге использования MIDL один из получившихся файлов будет являться заголовочным и содержать описание вашего интерфейса примерно следующего вида:

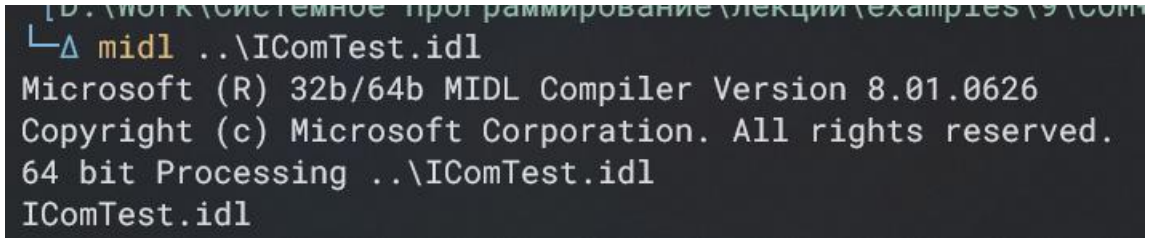
```
DECLSPEC_XFGVIRT(IUnknown, QueryInterface)
HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
    IComTest * This,
    /* [in] */ REFIID riid,
    /* [annotation][iid_is][out] */
    _COM_Outptr_  void **ppvObject);

DECLSPEC_XFGVIRT(IUnknown, AddRef)
ULONG ( STDMETHODCALLTYPE *AddRef )(
    IComTest * This);

DECLSPEC_XFGVIRT(IUnknown, Release)
ULONG ( STDMETHODCALLTYPE *Release )(
    IComTest * This);

DECLSPEC_XFGVIRT(IComTest, WhoAmI)
HRESULT ( STDMETHODCALLTYPE *WhoAmI )(
    IComTest * This,
```

Component Object Model

Но что такое этот MIDL?

MIDL (**MS Interface Definition Language**) – язык описания интерфейсов созданный Microsoft который позволяет программе или объекту, написанному на одном языке, взаимодействовать с другой программой, написанной на неизвестном ему языке

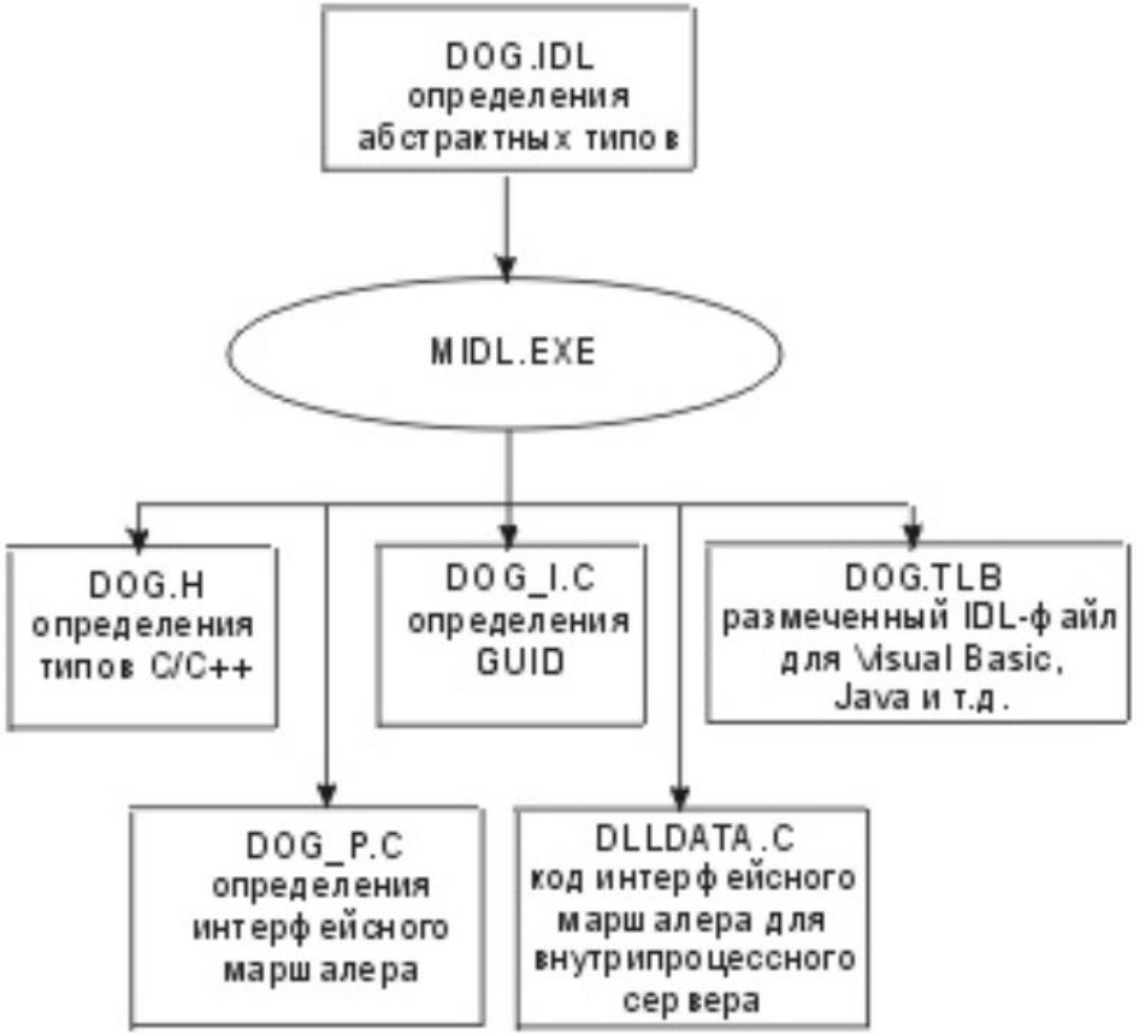
При создании COM-объектов на C++ использование MIDL является стандартной практикой, но не обязательной!



Component Object Model

IDL файлы содержат описание COM-компонентов и COM-интерфейсов не зависящим от языка способом

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

[
    object,
    uuid(C0C62619-3BC1-4095-9B9A-84503E37DAA5),
    version(1.0),
    helpstring("IComTest interface")
]
interface IComTest : IUnknown
{
    HRESULT WhoAmI([out] LPWSTR *pwszWhoAmI);
}
```



Component Object Model

Инструкция IDL **import** используется для ввода заголовочного файла (по сути аналогична директиве #include)
Атрибут **object** идентифицирует интерфейс как объектный и указывает компилятору MIDL генерировать код прокси/заглушки вместо клиентских и серверных заглушек RPC. Методы объектного интерфейса должны возвращать значение типа HRESULT
Атрибут **uuid** определяет идентификатор интерфейса (IID). Каждый интерфейс, класс и библиотека типов должны быть идентифицированы с помощью собственного уникального идентификатора (для получения такого значения можно воспользоваться Uuidgen.exe)

Component Object Model

Атрибут **version** определяет конкретную версию среди нескольких версий COM-интерфейса. С помощью атрибута version вы гарантируете, что для привязки разрешены только совместимые версии клиентского и серверного программного обеспечения
Атрибут **helpstring** определяет символьную строку, которая используется для описания элемента, к которому он применяется
Ключевое слово **interface** определяет имя интерфейса. Все интерфейсы объектов должны быть производными, прямо или косвенно, от Unknown

Component Object Model

Стоит наконец рассмотреть интерфейс **IUnknown**, который содержит следующие методы жизненного цикла:

- **AddRef** – увеличивает счётчик ссылок на интерфейс на 1
- **QueryInterface** – получение указателя на интерфейс по IID
- **Release** - уменьшает счётчик ссылок на интерфейс на 1

Данный «**счётчик ссылок на интерфейс**» необходим для отслеживания момента, когда экземпляр COM-компонента больше не требуется и может быть удалён

Все методы COM-интерфейса должны поддерживать соглашение о вызовах **stdcall**, а также возвращать **HRESULT** за исключением **AddRef** и **Release** – они возвращают текущее значение счётчика ссылок на интерфейс

Component Object Model

2. Создать COM-компонент путём написания класса который реализует ранее полученный COM-интерфейс

```
class CComServerTest : public IComTest
{
public:

    // IUnknown

    IFACEMETHODIMP_(ULONG) AddRef()
    {…
    }

    IFACEMETHODIMP_(ULONG) Release()
    {                                   …
    }

    IFACEMETHODIMP QueryInterface(__in REFIID riid, __out void **ppv)
    {…
    }

    // IComTest
    IFACEMETHODIMP WhoAmI(_Out_ LPWSTR *ppwszWhoAmI)
    {…
    }
```

```
// ANOTHER POSSIBLE IMPLEMENTATION FOR QueryInterface
HRESULT hr = (ppv != nullptr) ? S_OK : E_INVALIDARG;
if (SUCCEEDED(hr)) {
  *ppv = nullptr;
  hr = E_NOINTERFACE;

  if (__uuidof(IComTest) == riid) {
    *ppv = static_cast<IComTest *>(this);
    hr = S_OK;
  } else if (__uuidof(IUnknown) == riid) {
    *ppv = static_cast<IUnknown *>(this);
    hr = S_OK;
  }

  if (SUCCEEDED(hr)) {
    reinterpret_cast<IUnknown *>(*ppv)->AddRef();
  }
}
```

Component Object Model

3. Реализовать фабрику классов путём реализации стандартного COM-интерфейса **IClassFactory** для удобного управления жизненным циклом COM-компонентов

COM требует, чтобы каждый класс имел собственную фабрику классов для создания экземпляров, но многие классы фактически могут использовать одну и ту же реализацию фабрики классов

Component Object Model

COM-интерфейс **IClassFactory** предоставляет следующие методы:

- **CreateInstance** – метод предназначенный для создания экземпляра COM-компонента
- **LockServer** – увеличение счётчика блокировки COM-сервера

Блокировка COM-сервера предназначена для гарантии того, что он не будет закрыт раньше времени (DLL не будет выгружена)

```
MIDL_INTERFACE("00000001-0000-0000-C000-000000000046")
IClassFactory : public IUnknown
{
public:
    virtual /* [local] */ HRESULT STDMETHODCALLTYPE CreateInstance(
        /* [annotation][unique][in] */
        _In_opt_  IUnknown *pUnkOuter,
        /* [annotation][in] */
        _In_  REFIID riid,
        /* [annotation][iid_is][out] */
        _COM_Outptr_  void **ppvObject) = 0;

    virtual /* [local] */ HRESULT STDMETHODCALLTYPE LockServer(
        /* [in] */ BOOL fLock) = 0;

};
```

Component Object Model

Фабрика классов также помогает удобно следить за жизненным циклом COM-компонент

Это является важной частью работы COM-сервера, так как при попытке освободить его ресурсы, вывод о том можно это сделать или нет, основывается на том факте используются ли хоть какие-то его COM-компоненты или нет

Для этого на сервере существует такое понятие как «**счётчик экземпляров компонент**»

Увеличение этого счётчика происходит в конструкторе COM-компонента (он вызывается методом **CreateInstance**), а уменьшается в деструкторе

Component Object Model

4. Реализовать набор из 5 обязательных функций DLL которые обеспечивают работу одного или нескольких COM-компонентов и которые обязательно экспортируются из DLL

```
; ComSampleServer.def
LIBRARY       ComSampleServer.dll
EXPORTS
    DllGetClassObject   PRIVATE
    DllCanUnloadNow     PRIVATE
    DllRegisterServer   PRIVATE
    DllUnregisterServer PRIVATE
    DllMain             PRIVATE
```

Component Object Model

Название функции	Описание функции
DllCanUnloadNow	Функция автоматически вызывается OLE32.DLL перед попыткой клиентом выгрузить COM-сервер. В зависимости от результата работы функции OLE32.DLL выгружает или не выгружает COM-сервер
DllGetClassObject	Первая функция компонента, вызываемая OLE32.DLL при работе с клиентом. Функция проверяет идентификатор компонента, создает фабрику классов компонента и через параметры возвращает OLE32.DLL указатель на стандартный интерфейс IClassFactory
DllInstal	Функция вызывается утилитой regsvr32 при наличии соответствующего параметра, применяется для выполнения дополнительных действий при регистрации и удаления регистрации компонентов
DllRegisterServer	Функция вызывается утилитой regsvr32 при наличии со ответствующего параметра, применяется для регистрации компонентов сервера в реестре операционной системы
DllUnregisterServer	Функция вызывается утилитой regsvr32 при наличии соответствующего параметра, применяется для удаления информации о компонентах сервера из реестра операционной системы

Component Object Model

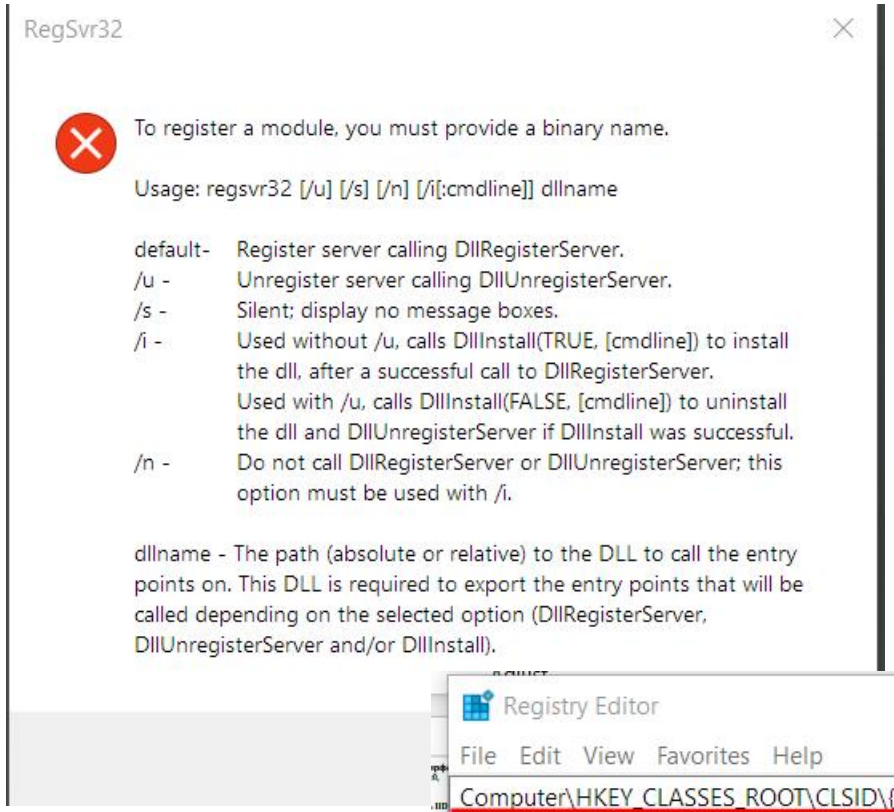
5. Скомпилировать COM-сервер

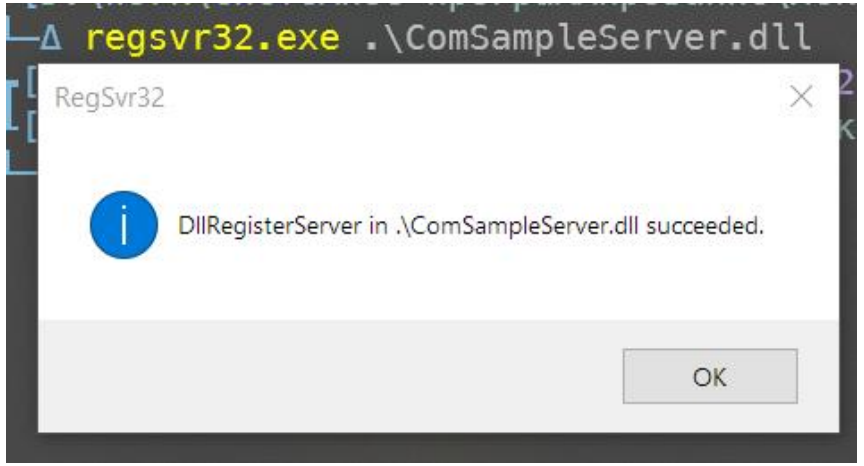
6. Зарегистрировать COM-сервер с использованием утилиты **regsvr32** (по сути данная утилита просто вызывает некоторые экспортируемые функции из DLL)

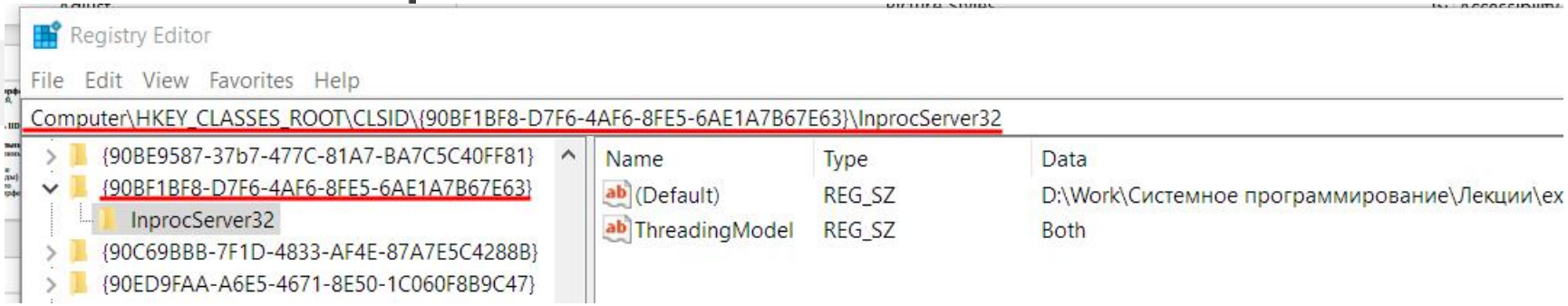
Что это и зачем?

Для реализации свойства «Независимости от местоположения» в Windows каждый COM-компонент должен быть зарегистрирован в Windows-реестре. Для регистрации компонента и применяется специальная утилита **regsvr32**

Component Object Model







Component Object Model

7. Разработать COM-клиент:

- Работа клиента должна начинаться с инициализации библиотеки OLE32 (вызов функции **CoInitializeEx**)
- Для создания экземпляра компонента необходимо вызвать функцию **CoCreanteInstance**
- Для получения указателя на другие интерфейсы можно применить метод **QueryInterface** стандартного интерфейса **IUnknown**
- Работа клиента должна завершаться освобождением библиотеки OLE32 (вызов функции **CoUninitialize**)

Component Object Model

Функции **CoInitialize** и **CoInitializeEx** инициализируют статические и загружаемые библиотеки COM, после чего могут использоваться остальные функции COM API (стоит отметить, что все этии функции имеют приставку **Co..**, например CoCreateInstance, CoGetClassObject и т.д.)

Первый параметр обеих функций зарезервирован и должен устанавливаться в **NULL\nullptr**

Разница функций лишь в том, что вторая функция позволяет выбирать различные потоковые модели COM

```
HRESULT CoInitializeEx(
  [in, optional] LPVOID pvReserved,
  [in]           DWORD  dwCoInit
);
```

Component Object Model

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();
```

Component Object Model

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, IUnknown) вместо указателя на IClassFactory

```
HRESULT CoCreateInstance(
  [in]  REFCLSID  rclsid,
  [in]  LPUNKNOWN pUnkOuter,
  [in]  DWORD     dwClsContext,
  [in]  REFIID    riid,
  [out] LPVOID    *ppv
);
```

```
HRESULT CoGetClassObject(
  [in]                REFCLSID rclsid,
  [in]                DWORD    dwClsContext,
  [in, optional] LPVOID   pvReserved,
  [in]                REFIID   riid,
  [out]               LPVOID   *ppv
);
```

Component Object Model

Параметр ***rclsid*** – ссылка на CLSID для заданного компонента

Параметр ***pUnkOuter*** – используется при агрегации, в остальном должен быть NULL

Параметр ***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

Параметр ***riid*** – ссылка на IID для заданного интерфейса, который необходимо возвратить из созданного компонентного объекта

Параметр ***ppvObj*** – указатель типа void* на возвращаемый интерфейс

Component Object Model

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**

```
void CoUninitialize();
```

Component Object Model

При этом при работе с COM-компонентом **клиент должен «знать»** только GUID-идентификатор этого компонента (**CLSID**), GUID идентификаторы (**IID**), тип сервра и структуры (**сигнатуры соответствующих методов**) произвольных интерфейсов компонента, которые он предполагает применять

Жизненный цикл COM-сервера:

- Не может быть выгружен пока счётчик экземпляров компонент не равен нулю (экземпляр COM-компоненты обычно уникален в рамках одного процесса, т.е. по сути Singleton на уровне процесса)
- Экземпляр COM-компоненты не может быть выгружен пока счётчик ссылок на интерфейсы не равен нулю
- Не может быть выгружен пока счётчик блокировок (**LockServer**) не равен нулю

Component Object Model

Кроме этого, стоит обратить внимание на работу с памятью в COM-приложениях

COM определяет пару функций для выделения и освобождения памяти в куче:

- Функция **CoTaskMemAlloc** выделяет блок памяти
- Функция **CoTaskMemFree** освобождает блок памяти, который был выделен с помощью **CoTaskMemAlloc**

Почему COM определяет свои собственные функции выделения памяти? Одна из причин заключается в том, чтобы обеспечить уровень абстракции над распределителем кучи

Component Object Model

В противном случае некоторые методы могли бы вызывать malloc, а другие - new. Тогда вашей программе пришлось бы вызывать free в одних случаях и delete в других, и отслеживать все это быстро стало бы невозможно

Функции выделения памяти COM создают единый подход

Еще одним соображением является тот факт, что COM – это двоичный стандарт, поэтому он не привязан к определенному языку программирования. Следовательно, COM не может полагаться на какую-либо специфичную для языка форму распределения памяти

Также существуют некоторые «лучшие практики» при работе с COM: **тут**

Component Object Model

Для размещения компонентов в Windows могут быть применены **два вида контейнеров**: DLL-файл и EXE-файл

Приложения, использующие COM-компоненты (вызывающие функции интерфейсов, реализованных COM-компонентами), называют **COM-клиентами**, а контейнеры с расположенными в них компонентами – **COM-серверами**

В зависимости от типа контейнера и места его расположения (локальное или удаленное) различают несколько типов серверов: **INPROC** (DLL, локальный), **LOCAL** (EXE, локальный), **REMOTE** (EXE, удаленный)

Component Object Model

COM-серверы в зависимости от количества реализуемых ими компонентов подразделяются на «**однокомпонентные**» и «**многокомпонентные**»

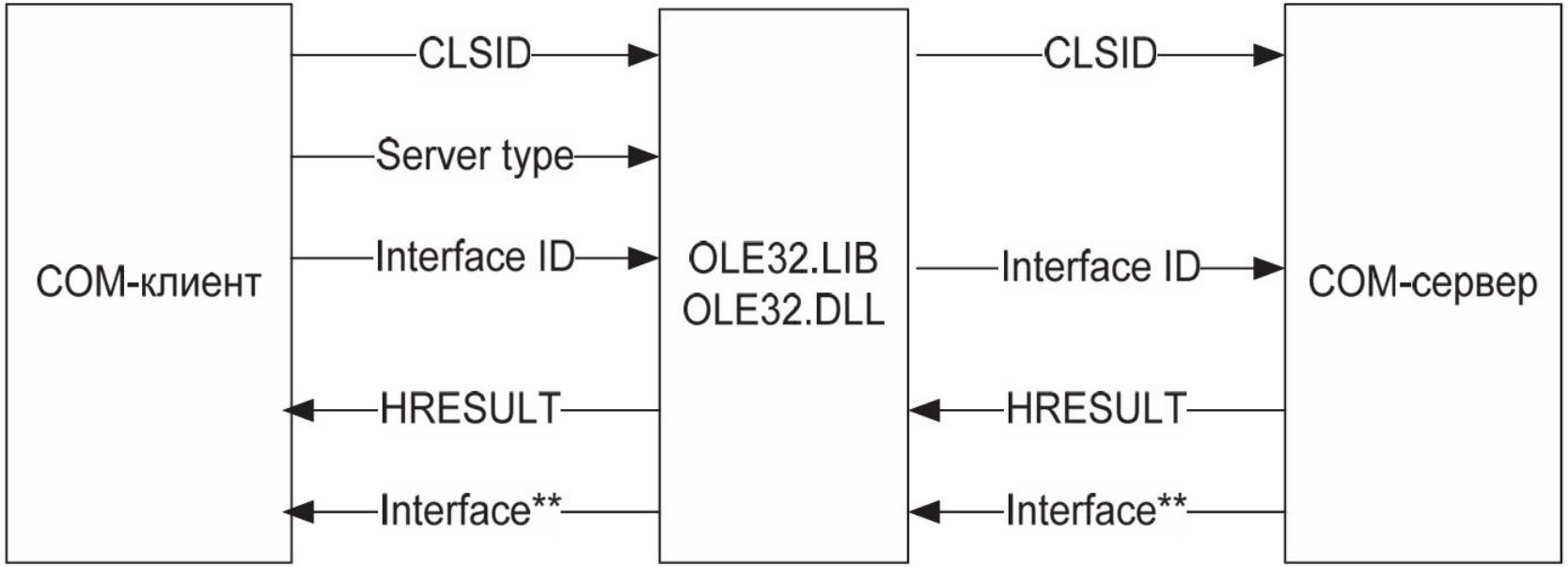
Соответственно, если в контейнере расположен только один компонент, то сервер – «однокомпонентный», если два и более – «многокомпонентный»

При этом COM-сервер сам может выступать в виде клиента, если он вызывает методы интерфейсов, реализованные другими компонентами

Component Object Model

Принципы взаимодействия клиента и сервера

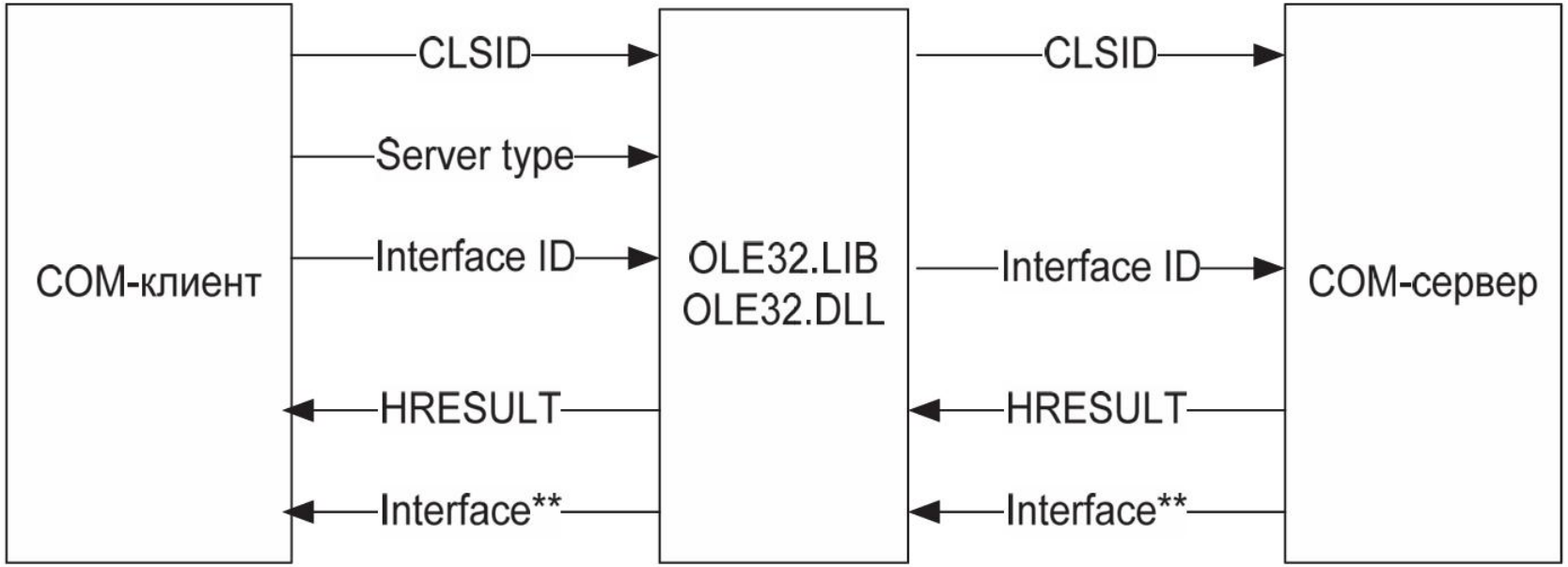
Поддержка программ соответствующих COM-модели в операционной системе Windows обеспечивается с помощью динамически подключаемой библиотеки **OLE32.DLL** и соответствующей ей библиотеки экспорта функций **OLE32.LIB**



Component Object Model

Принципы взаимодействия клиента и сервера

Именно OLE32.DLL по идентификатору CLSID через реестр операционной системы определяет место расположения контейнера компонента, загружает и инициализирует его



Component Object Model

Принципы взаимодействия клиента и сервера

За небольшим исключением все функции компонента должны возвращать результат в виде значения **HRESULT**, которое имеет следующую структуру:

```
//
//  Values are 32 bit values laid out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
```

Component Object Model

Принципы взаимодействия клиента и сервера

30-31 биты HRESULT отображают успешность выполнения функции COM-компонента
29 бит HRESULT отображает кем был определен данный статус код: пользователем или системой
28 бит HRESULT является зарезервированным

```
//
//      Sev - is the severity code
//
//          00 - Success
//          01 - Informational
//          10 - Warning
//          11 - Error
//
//      C - is the Customer code flag
//
//      R - is a reserved bit
//
//      Facility - is the facility code
//
//      Code - is the facility's status code
//
```

Component Object Model

Принципы взаимодействия клиента и сервера

16-27 биты HRESULT отображают к какой технологии относится статус код

0-15 биты HRESULT отображают точный результат в рамках заданной технологии и серьезности

```
//
#define FACILITY_NULL                    0
#define FACILITY_RPC                     1
#define FACILITY_DISPATCH                2
#define FACILITY_STORAGE                 3
#define FACILITY_ITF                     4
#define FACILITY_WIN32                   7
#define FACILITY_WINDOWS                 8
#define FACILITY_SSPI                    9
#define FACILITY_SECURITY                9
#define FACILITY_CONTROL                 10
#define FACILITY_CERT                    11
#define FACILITY_INTERNET                12
#define FACILITY_MEDIASERVER             13
#define FACILITY_MSMQ                    14
#define FACILITY_SETUPAPI                15
#define FACILITY_SCARD                   16
```

Component Object Model

Принципы взаимодействия клиента и сервера

Чтобы определить, был ли вызов успешен или произошла ошибка, можно воспользоваться макросами: **SUCCEEDED**() – успех и **FAILED**() – неудача

```
hr = pComTest->WhoAmI(pwszWhoAmI: &pwszWhoAmI);
if (SUCCEEDED(hr))
```

Component Object Model

Существует два основных типа серверов: **in-process** (в процессе) и **out-of-process** (вне процесса)

Серверы **in-process** реализуются в динамической библиотеке (DLL), а серверы **out-of-process** реализуются в исполняем файле (EXE)

Серверы **out-of-process** могут размещаться либо на локальном компьютере, либо на удаленном компьютере

Кроме того, COM предоставляет механизм, который позволяет серверу **in-process** (DLL) запускаться в суррогатном процессе EXE, чтобы получить преимущество выполнения процесса на удаленном компьютере

Component Object Model

Построение **in-process** и **out-of-process** серверов ничем не отличается с точки зрения структуры, однако при работе с **out-of-process** серверами возникает некоторая сложность, а именно: как получить указатель на функцию или объект которые располагается в другом процессе

В таком случае между клиентом и сервером появляется прослойка в виде прокси-объекта

Для создания таких объектов необходимо будет применять MIDL

Системное программирование

Лекция 5

Component Object Model