# Про регистрацию

Представьте, что у вас есть программа (клиент), которая хочет использовать функциональность, предоставляемую другим компонентом (COM-сервером). Как клиент узнает:

* Существует ли вообще такой компонент?
* Где находится файл (DLL или EXE), реализующий этот компонент?
* Как создать экземпляр этого компонента?

COM решает эту проблему с помощью **централизованного хранилища информации** - **Реестра Windows**. Регистрация - это процесс записи необходимой информации о COM-сервере в этот реестр. Когда клиент хочет использовать COM-компонент, он обращается к COM-подсистеме Windows, которая ищет информацию в реестре и на основе найденного создает и предоставляет клиенту экземпляр сервера.

В реестре создаются специальные записи, которые описывают COM-сервер. Самые важные из них:

* **CLSID (Class Identifier):** Уникальный глобальный идентификатор (GUID) для *каждого* COM-класса (компонента), который может создавать сервер. Это как уникальный номер телефона компонента. Пример: {ABCDEF12-3456-7890-ABCD-EF1234567890}. Клиент обычно использует CLSID, чтобы запросить создание объекта.
  + Хранится в **HKEY\_CLASSES\_ROOT**\CLSID\{GUID}.
* **ProgID (Programmatic Identifier):** Человекочитаемое имя для COM-класса. Например, Excel.Application или MyProject.MyComponent. ProgID удобнее использовать в скриптах или при позднем связывании. ProgID обычно ссылается на соответствующий CLSID.
  + Хранится в HKEY\_CLASSES\_ROOT\ProgID (например, HKEY\_CLASSES\_ROOT\Excel.Application).
  + Внутри ветки ProgID есть ключ CLSID, указывающий на соответствующий GUID.
* **Путь к файлу сервера:** Самое важное для понимания различий DLL/EXE!
  + Для **DLL-серверов (In-Process)**: Регистрируется полный путь к DLL-файлу в под-ключе **InprocServer32 внутри ветки CLSID**. Например: HKEY\_CLASSES\_ROOT\CLSID\{GUID}\InprocServer32 -> (По умолчанию) = C:\path\to\myserver.dll. Это говорит COM, что сервер нужно загружать *в то же самое адресное пространство (процесс)*, что и клиент.
  + Для **EXE-серверов (Out-of-Process)**: Регистрируется полный путь к EXE-файлу в под-ключе **LocalServer32 внутри ветки CLSID**. Например: HKEY\_CLASSES\_ROOT\CLSID\{GUID}\LocalServer32 -> (По умолчанию) = C:\path\to\myserver.exe. Это говорит COM, что сервер нужно запускать *в отдельном процессе*.
* **Threading Model (Модель потоков):** (В основном для DLL) Указывает, как сервер работает с потоками (Apartment, Free, Both, Neutral). Хранится в **InprocServer32**.
* **Type Library (Библиотека типов):** (Необязательно, но очень полезно) Информация об интерфейсах, методах, свойствах, событиях, которые предоставляет сервер. Позволяет клиентам (особенно при разработке) понимать, как взаимодействовать с сервером. Ссылка на библиотеку типов (TLB-файл или ресурс внутри DLL/EXE) хранится в ключе TypeLib внутри ветки CLSID и в HKEY\_CLASSES\_ROOT\TypeLib.

**А) Регистрация DLL-серверов (In-Process)**

1. **Код регистрации внутри DLL:** Разработчик DLL-сервера обязан реализовать и экспортировать две стандартные функции:
   * **DllRegisterServer():** Эта функция содержит код, который создает все необходимые записи в реестре Windows (CLSID, ProgID, InprocServer32 с путем к этой DLL, модель потоков, TypeLib и т.д.).
   * **DllUnregisterServer():** Эта функция содержит код, который удаляет *все* записи, созданные DllRegisterServer(). Это нужно для чистого удаления компонента из системы.
2. **Как вызываются эти функции?** С помощью специальной утилиты Windows - **regsvr32.exe**.
   * **Регистрация:** Вы запускаете в командной строке (часто с правами администратора):

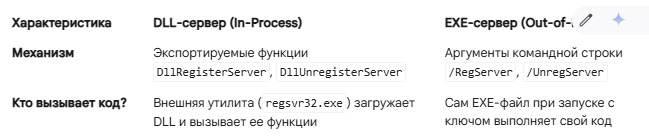
regsvr32.exe C:\path\to\myserver.dll

Что делает regsvr32.exe?

* + - Загружает указанную DLL в свое собственное адресное пространство.
    - Находит адрес экспортируемой функции DllRegisterServer внутри DLL.
    - Вызывает эту функцию DllRegisterServer().
    - DLL выполняет свой код регистрации (запись в реестр).
    - regsvr32.exe выгружает DLL и сообщает об успехе или неудаче.
  + **Отмена регистрации:**

regsvr32.exe /u C:\path\to\myserver.dll

regsvr32.exe делает то же самое, но вызывает функцию DllUnregisterServer().



* **GUID/UUID:** Сам уникальный номер (128 бит).
* **IID:** GUID/UUID, играющий роль идентификатора **интерфейса**.
* **CLSID:** GUID/UUID, играющий роль идентификатора **класса** (компонента).

**Типы данных:**

**GUID**:

* + Это **основная структура данных** для хранения 128-битного уникального идентификатора.
  + Определена в заголовочных файлах Windows (например, guiddef.h).

1. **IID**:
   * Это **псевдоним (typedef)** для типа GUID.
   * Определен как: typedef GUID IID;
   * Используется **семантически**, чтобы показать, что данный GUID используется как *идентификатор интерфейса*. Технически, это та же самая структура GUID.
2. **CLSID**:
   * Это **псевдоним (typedef)** для типа GUID.
   * Определен как: typedef GUID CLSID;
   * Используется **семантически**, чтобы показать, что данный GUID используется как *идентификатор класса (компонента)*. Технически, это та же самая структура GUID.
3. **REFIID**:
   * Это **псевдоним (typedef)** для **константной ссылки** на IID (а значит, и на GUID).
   * Определен как: typedef const IID &REFIID; (или эквивалентно typedef const GUID &REFIID;)
   * Используется в качестве **типа для параметров функций** (как riid в QueryInterface), которые принимают идентификатор интерфейса. Использование ссылки (&) эффективнее, чем копирование всей 16-байтной структуры GUID. const гарантирует, что функция не изменит переданный идентификатор.
4. **REFCLSID**:
   * Это **псевдоним (typedef)** для **константной ссылки** на CLSID (а значит, и на GUID).
   * Определен как: typedef const CLSID &REFCLSID; (или эквивалентно typedef const GUID &REFCLSID;)
   * Используется в качестве **типа для параметров функций** (как rclsid в CoCreateInstance), которые принимают идентификатор класса.
5. **REFGUID**:
   * Это **псевдоним (typedef)** для **константной ссылки** на GUID.
   * Определен как: typedef const GUID &REFGUID;
   * Используется как общий тип для параметров функций, принимающих любой GUID по ссылке.

При разработке повторно используемого ПО системный программист использует существующие или разрабатывает новые соглашения, которым оно должно соответствовать. Эти соглашения оформляются как спецификации или корпоративные стандарты и определяют, например, правила именования, типы данных и структуру интерфейсов. Пример — спецификация COM (Component Object Model) от Microsoft.

Component Object Model (COM) — объектная модель компонентов от Microsoft, предназначенная для проектирования и создания модулей, независимых от языка программирования. COM реализована во всех версиях Windows, а в других системах (например, UNIX или macOS) может поддерживаться сторонними средствами. Модель развивалась как продолжение подходов к модульной разработке на C++.

Одна из ключевых особенностей COM — **это предоставление общего двоичного стандарта для компонентов**. Благодаря ему модули, созданные на разных языках программирования и под разные операционные системы, могут взаимодействовать без перекомпиляции или изменения кода. Это особенно важно для повторного использования программного обеспечения в разных средах.

Компоненты можно распространять в виде **двоичных файлов** (например, DLL или EXE), и они будут работать независимо от языка или системы разработки.

Другое важное свойство СОМ известно под названием ***независимости от местоположения***. Независимость от местоположения означает, что пользователь компонента, клиент, не обязательно должен знать, где находится определенный компонент

Компонент может находиться непосредственно в адресном пространстве задачи клиента (DLL-файл), в пространстве другой задачи на том же компьютере (ЕХЕ-файл) или на компьютере, расположенном за сотни миль (распределенный объект).

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

Ключевая идея – **разделение интерфейса и реализации**. Клиент работает только с **интерфейсами**, не зная деталей реализации объекта.

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

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

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

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

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

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

**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**

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

**Vtable** – это, по сути, таблица (массив) указателей на реальные реализации виртуальных методов класса. Когда вы вызываете виртуальный метод через указатель на базовый класс, программа смотрит в Vtable объекта, на который указывает указатель, находит адрес нужного метода и вызывает его. Это позволяет вызывать правильную реализацию метода для объекта производного класса через указатель базового класса.

COM берет эту идею, но делает ее **стандартом**, не зависящим от конкретного языка программирования (как C++) или компилятора.

Когда вы работаете с COM-объектом, вы делаете это через **указатель на интерфейс. Этот адрес указывает на начало той части экземпляра COM-объекта, которая соответствует данному интерфейсу.**

По адресу, на который указывает ваш интерфейсный указатель, самое первое, что там лежит (первые 4 или 8 байт, в зависимости от архитектуры x86/x64) – это **другой указатель**. Этот *другой* указатель указывает на **Vtable**, ассоциированную с этим конкретным интерфейсом для этого конкретного объекта (**V-table создается для КЛАССА**).

Функции в Vtable расположены не хаотично. Их порядок строго соответствует порядку объявления методов в определении интерфейса.

Поэтому доступ к методам интерфейса идет через эту двойную косвенность: Объект -> vptr -> Функция.

Компонент *обязан* реализовать базовый интерфейс IUnknown. IUnknown предоставляет **фундаментальные механизмы**, без которых взаимодействие между компонентами было бы невозможно:

* **Единое Управление Временем Жизни:** У конкретного экземпляра COM-объекта есть только **один** счетчик ссылок, независимо от того, сколько разных интерфейсов он реализует.
  + Вызов AddRef через *любой* указатель на интерфейс этого объекта (pMath->AddRef(), pPersist->AddRef()) увеличивает **этот единственный** счетчик.
  + Вызов Release через *любой* указатель на интерфейс этого объекта (pMath->Release(), pPersist->Release()) уменьшает **этот единственный** счетчик.
  + Объект уничтожается только тогда, когда **этот единственный** счетчик достигает нуля, независимо от того, какой интерфейсный указатель был последним, для которого вызвали Release.
* **Полная Доступность через QueryInterface:** Вы можете вызвать QueryInterface через *любой* имеющийся у вас указатель на интерфейс данного объекта, чтобы запросить *любой другой* интерфейс, реализованный этим же объектом.
  + QueryInterface должен корректно работать независимо от того, через какой интерфейс его вызвали, и всегда ссылаться на **общую** таблицу поддерживаемых интерфейсов объекта.

**Несмотря на множественное наследование интерфейсов, сам COM-объект должен иметь единую, общую реализацию методов IUnknown (QueryInterface, AddRef, Release) и один общий счетчик ссылок (m\_lRef).**

IUnknown предоставляет три фундаментальных метода, необходимых для решения ключевых проблем взаимодействия:

* QueryInterface: Для обнаружения других интерфейсов (функциональности) компонента. Позволяет клиенту спросить у объекта: "Поддерживаешь ли ты интерфейс с идентификатором riid?". Если да, то метод должен вернуть указатель на этот интерфейс.
* AddRef и Release: Для управления временем жизни компонента.

**Стандартный Тип Возвращаемого Значения: HRESULT** — это **стандартный способ** для большинства функций COM и многих функций Windows API сообщить программе, которая их вызвала, **успешно** ли выполнилась операция или **произошла ошибка**.

* **Старший бит (бит 31): Флаг Серьезности (Severity)**
  + Это **самый важный** бит. Он определяет общий результат:
    - 0 (ноль) означает **УСПЕХ** (Success).
    - 1 (единица) означает **ОШИБКУ** (Failure/Error).
* **Следующие 15 битов (биты 30-16): Код "Средства" или "Источника" (Facility Code)**
  + Это позволяет **группировать** коды ошибок по их происхождению. Программист или система могут понять, относится ли ошибка, например, к проблемам с памятью, к сетевому взаимодействию или к неправильному использованию COM-интерфейса.
* **Младшие 16 битов (биты 15-0): Код Статуса (Status Code)**
  + Эти биты предоставляют **конкретную информацию** о том, что произошло.
  + Если старший бит = 0 (успех), эти биты могут уточнять *тип* успеха (например, S\_OK - всё отлично, S\_FALSE - операция выполнена, но логический результат "ложь").
  + Если старший бит = 1 (ошибка), эти биты указывают на **конкретную причину ошибки** (например, "интерфейс не найден", "недостаточно памяти", "неверный аргумент").

Поскольку структура HRESULT довольно сложна, напрямую анализировать биты неудобно. Поэтому существуют стандартные **макросы** (предопределенные команды препроцессора C/C++), которые упрощают проверку:

* **SUCCEEDED(hresult\_value):**
  + Этот макрос принимает значение HRESULT в качестве аргумента.
  + Он возвращает TRUE (истину), если старший бит hresult\_value равен **0** (т.е. операция завершилась успешно, даже если это был успех типа S\_FALSE).
  + Он возвращает FALSE (ложь), если старший бит равен **1** (произошла ошибка).
  + **Это основной способ проверить, успешно ли выполнился вызов COM-метода.**
* **FAILED(hresult\_value):**
  + Этот макрос делает обратное.
  + Он возвращает TRUE (истину), если старший бит hresult\_value равен **1** (произошла ошибка).
  + Он возвращает FALSE (ложь), если старший бит равен **0** (операция завершилась успешно).
  + **Это основной способ проверить, произошла ли ошибка при вызове COM-метода.**

Возвращаемые значения в Win32 предваряются префиксом S\_ в случае нормального завершения и префиксом Е\_ — в случае ошибки.

**Суть QueryInterface:**

*Любой* указатель на COM-интерфейс будет иметь метод QueryInterface.

**REFIID riid**: В сигнатуре функции QueryInterface(REFIID riid, void\*\* ppv) первый параметр riid – это ссылка на этот самый GUID/IID, который указывает, какой интерфейс ты запрашиваешь.

**Если QueryInterface выполняется успешно**, метод записывает по адресу, переданному в ppvObject, **валидный указатель** на запрошенный интерфейс.

* Сам COM-объект берет на себя ответственность за управление своим временем жизни. Он содержит **внутренний счетчик ссылок**.

**Метод AddRef():**

* **Назначение:** Увеличить внутренний счетчик ссылок на единицу.
* **Когда вызывается:** Каждый раз, когда клиент получает **новый** указатель на интерфейс этого объекта. Это происходит в двух основных случаях:
  + При успешном вызове QueryInterface().
  + Когда клиент сам **копирует** существующий указатель на интерфейс и собирается использовать эту копию независимо (например, сохраняет ее в переменную-член другого объекта или передает в другую функцию/поток, которая будет владеть этой копией).

**Метод Release():**

* + **Назначение:** Уменьшить внутренний счетчик ссылок на единицу и проверить, не пора ли объекту самоуничтожиться.
  + **Когда вызывается:** Каждый раз, когда клиент **завершает использование** указателя на интерфейс, который он ранее получил (и для которого был сделан AddRef).
* Клиент **обязан** вызывать Release для каждого указателя, которым он владеет, когда он ему больше не нужен.
* Клиент **обязан** вызывать AddRef, если он делает копию указателя для долгосрочного хранения.

Программист, создающий COM-компонент (например, класс Math):

* Обязан: Реализовать методы AddRef, Release и QueryInterface.
* Обязан: Внутри класса завести переменную для хранения счетчика ссылок (как m\_lRef в примере).
* Обязан: В AddRef увеличивать этот счетчик.
* Обязан: В Release уменьшать этот счетчик и проверять, не стал ли он равен нулю. Если стал – вызывать delete this; (самоуничтожение).
* Обязан: В QueryInterface при успешном возврате указателя вызывать AddRef.

Сложность множества интерфейсов

* поскольку СОМ-интерфейсы фактически являются указателями на виртуальную таблицу функций C++, класс с множеством интерфейсов требует создания множества виртуальных таблиц.
* необходимость подсчета количества обращений. Все интерфейсы СОМ-объекта должны взаимодействовать между собой в обеспечение подсчета обращений. Для каждого объекта существует только один счетчик обращений, и интерфейсы должны использовать его совместно.



* **COM-компонент (или COM-класс):** Это реализация одного или нескольких COM-интерфейсов. По сути, это код, выполняющий определенную логику и предоставляющий доступ к ней через стандартные интерфейсы (IUnknown и другие).
* **COM-сервер:** Это исполняемый модуль (файл .dll или .exe), который содержит код одного или нескольких COM-компонентов и необходимую инфраструктуру для их создания и предоставления клиентам.

Классификация COM-серверов основана на том, где выполняется код компонента относительно процесса, который его использует (клиента).

**1. Внутризадачный сервер (In-Process Server, InProc)**

* **Физическое представление:** Всегда **DLL** (динамически подключаемая библиотека).
* **Контекст выполнения:** Код DLL-сервера **загружается непосредственно в адресное пространство вызывающего процесса-клиента**. Клиент и компонент выполняются в рамках одного и того же процесса Windows.
* **Взаимодействие:**
  + **Вызов метода:** Клиент получает **прямой указатель** на объект в памяти. Вызов метода — это обычный **прямой вызов функции** внутри этого же процесса (через v-table). Никаких границ пересекать не нужно.
  + **Передача данных:** Указатели на данные можно передавать напрямую, так как клиент и компонент видят **одну и ту же память**.

**2. Локальный сервер (Local Server, Out-of-Process)**

* **Физическое представление:** Обычно **EXE** (исполняемый файл).
* **Контекст выполнения:** Сервер запускается как **отдельный, независимый процесс** на **той же самой машине**, что и клиент. Клиент и компонент выполняются в разных адресных пространствах.
* **Взаимодействие:**
  + Прямой вызов методов невозможен из-за изоляции адресных пространств. COM использует механизм **межпроцессного взаимодействия (Inter-Process Communication, IPC)**, часто основанный на легковесной версии RPC (Remote Procedure Call).
  + В процессе клиента создается **прокси-объект (proxy)**. Этот объект реализует тот же COM-интерфейс, что и реальный компонент, но его методы не содержат логики компонента. Клиент получает указатель на этот прокси и вызывает его методы.
  + В процессе сервера создается **заглушка (stub)**.
  + Когда клиент вызывает метод прокси, прокси **маршалит** параметры вызова (упаковывает их в не зависящий от процесса формат), передает их через механизм IPC операционной системы заглушке в процессе сервера.
  + Заглушка получает упакованные данные, **демаршалит** их (распаковывает) и вызывает соответствующий метод реального объекта компонента в процессе сервера.
  + Возвращаемые значения проходят обратный путь: от компонента к заглушке, маршалинг, передача через IPC, демаршалинг прокси и возврат клиенту.

**3. Удаленный сервер (Remote Server)**

* **Физическое представление:** Обычно **EXE** (или DLL, размещенная в суррогатном процессе) на **другом компьютере**.
* **Контекст выполнения:** Сервер выполняется в **собственном процессе** на **удаленной машине**, доступной по сети.
* **Взаимодействие:**
  + Механизм аналогичен локальному серверу (прокси/заглушка, маршалинг/демаршалинг), но IPC осуществляется через **сетевые протоколы** (например, RPC поверх TCP/IP). Эта часть COM называется DCOM (Distributed COM).

1. **Раздельная Компиляция и Сокрытие Реализации:**
   * Когда вы компилируете клиентскую программу, она знает только об **интерфейсах** компонента (например, IMath из IMath\_h.h, сгенерированного MIDL). Она **не знает** и **не должна знать** о конкретном имени класса реализации внутри DLL/EXE (например, CMathImplementation).
   * Оператор new требует, чтобы компилятор знал **полное определение класса** (его размер, конструкторы и т.д.) во время компиляции клиента. Если бы это было так, клиент стал бы жестко привязан к деталям реализации компонента, что противоречит идее COM о разделении интерфейса и реализации.
2. **Языковая Независимость:**
   * Оператор new (или его аналоги) **специфичен для конкретного языка** (C++, C#, Java и т.д.).
   * Клиент может быть написан на Visual Basic, Python или C#, а компонент — на C++. Как клиент на Python вызовет конструктор C++? COM должен предоставлять **стандартный, не зависящий от языка** механизм создания объектов.
3. **Процессные Границы (для EXE-серверов):**
   * Если COM-компонент находится в отдельном **EXE-файле**, он работает в **другом адресном пространстве**, чем клиент.
   * Оператор new работает **только в пределах текущего процесса**. Он не может выделить память и вызвать конструктор в совершенно другом процессе. Операционная система этого не позволит из соображений безопасности и стабильности.
4. **Сетевая Прозрачность (для Remote-серверов):**
   * Если компонент находится на **другом компьютере**, new тем более не применим. Невозможно напрямую создать объект на удаленной машине с помощью локального оператора new.

**Решение: Интерфейс IClassFactory:**

COM определяет стандартный интерфейс IClassFactory. Единственная задача объекта, реализующего IClassFactory, – это **создавать экземпляры *другого*, связанного с ним COM-компонента**.

**Это тоже COM-объект:** Фабрика классов сама является COM-объектом, поэтому она тоже должна реализовывать IUnknown (QueryInterface, AddRef, Release).

Фабрика сразу возвращает указатель на один из интерфейсов компонента!

1. HRESULT **CreateInstance**(LPUNKNOWN pUnkOuter, REFIID riid, void\*\* ppv):
   * **Главный метод.** Вызывается клиентом, когда он хочет создать новый экземпляр "основного" компонента (например, Math).
   * pUnkOuter: Используется для *агрегации* (более сложный механизм COM, часто равен NULL).
   * riid: Идентификатор (IID) **интерфейса**, который клиент хочет получить от **только что созданного** объекта. Это оптимизация, чтобы не делать сразу после создания еще один QueryInterface.
   * ppv: Выходной параметр, куда будет помещен указатель на запрошенный интерфейс (riid) **нового** объекта (Math).
2. HRESULT **LockServer**(BOOL fLock):
   * Метод LockServer позволяет клиенту **временно "заблокировать" сервер в памяти**, даже если клиент освободил все *экземпляры* объектов, созданные этим сервером. Клиент как бы говорит серверу: "Пожалуйста, не выгружайся/не завершай работу пока, ты мне скоро снова понадобишься!".
   * Если fLock равно TRUE, сервер не должен выгружаться из памяти, даже если нет активных объектов. Если FALSE, блокировка снимается.

**Что такое COM Apartments (Квартиры)?**

**COM Apartments:** Это механизм (**наборы правил**) внутри процесса для управления доступом потоков к COM-объектам и необходимостью маршалинга (упаковки вызовов).

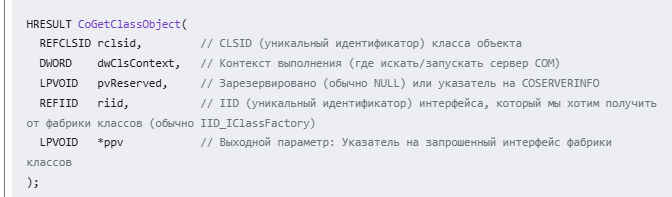
Существуют два основных типа апартаментов:

1. **STA (Single-Threaded Apartment):**
   * **Один поток** владеет STA. В процессе может быть много STA.
   * Объекты, созданные в STA, принадлежат этому потоку.
   * Все вызовы методов объекта **должны выполняться** в потоке-владельце STA.
   * Вызовы из *других* потоков **маршалятся** через очередь сообщений потока STA (требуется цикл обработки сообщений).
   * COM **обеспечивает сериализацию** вызовов (неявную синхронизацию). Компонент обычно **не требует** ручной синхронизации.
2. **MTA (Multi-Threaded Apartment):**
   * **Много потоков** могут принадлежать одной MTA. В процессе **только одна** MTA.
   * Объекты, созданные в MTA, доступны всем потокам MTA.
   * Вызовы методов от *любого* потока MTA к *любому* объекту MTA выполняются **напрямую** (без маршалинга).
   * COM **не обеспечивает синхронизацию** внутри MTA. Компонент **обязан быть потокобезопасным** (разработчик отвечает за синхронизацию).
   * Вызовы из STA в MTA маршалятся.

CoInitialize **инициализирует COM-библиотеку** для текущего потока, позволяя ему использовать COM-функции

1. **CoInitialize(NULL):**
   * Всегда инициализирует поток в **Single-Threaded Apartment (STA)**.
   * Более старая функция.
2. **CoInitializeEx(NULL, dwCoInit):**
   * Позволяет **явно выбрать** потоковую модель через второй параметр dwCoInit:
     + COINIT\_APARTMENTTHREADED: Инициализирует как **STA**.
     + COINIT\_MULTITHREADED: Инициализирует как **MTA** (Multi-Threaded Apartment).
     + (Есть и другие флаги, но эти основные).
   * Более новая и **предпочтительная** функция, так как дает больше контроля.

**Получение Фабрики Классов (CoGetClassObject)**



Клиент сначала запрашивает у COM специальный объект – **Фабрику Классов** (IClassFactory) для нужного компонента.

**Что делает:** Принимает **CLSID** компонента (уникальный ID, например CLSID\_Math), который клиент хочет создать.

Принимает **контекст** (CLSCTX\_INPROC\_SERVER для DLL, CLSCTX\_LOCAL\_SERVER для EXE на той же машине и т.д.), указывающий, где система должна искать и активировать COM-сервер, реализующий этот класс.

Принимает **IID** интерфейса, который клиент хочет получить от *фабрики* (почти всегда IID\_IClassFactory).

**Под капотом:** COM-библиотека использует CLSID и контекст для поиска информации **в реестре Windows**. Находит путь к DLL или EXE.

Если это DLL (CLSCTX\_INPROC\_SERVER): COM загружает эту DLL в процесс клиента и вызывает ее экспортируемую функцию DllGetClassObject, чтобы получить фабрику.

Если это EXE (CLSCTX\_LOCAL\_SERVER): COM запускает EXE (если он еще не запущен) и связывается с фабрикой, которую EXE должен был зарегистрировать с помощью CoRegisterClassObject.

Возвращает клиенту указатель на запрошенный интерфейс фабрики (например, IClassFactory\* pCF).

**3. Создание Экземпляра Компонента (IClassFactory::CreateInstance)**

Теперь у клиента есть фабрика (pCF). Он использует ее, чтобы наконец-то создать сам объект Math.

**4. Освобождение Фабрики (IClassFactory::Release)**

**5. Использование Компонента и Освобождение (IMath::Add, IMath::Release)**

**6. Деинициализация COM (CoUninitialize)**

**UUID/GUID – это 128-битное число.** Как достигается уникальность (упрощенно): Алгоритм генерации обычно использует комбинацию текущего времени с высоким разрешением, MAC-адреса сетевой карты компьютера (который предполагается уникальным) и/или других случайных или последовательных данных. Это гарантирует "статистическую уникальность" по всему миру и во времени.

**Макрос DEFINE\_GUID:** Это стандартный способ объявления константы GUID в коде.

**Как Получить GUID:** Разработчики не придумывают GUID вручную. Они используют специальные утилиты:

* + **uuidgen.exe:** Утилита командной строки, генерирует GUID. Можно запросить сразу несколько (-n50).
  + **guidgen.exe:** Графическая утилита (показана на рис. 5). Позволяет сгенерировать GUID и сразу скопировать его в буфер обмена в нужном формате (например, готовый DEFINE\_GUID(...)).
  + **Интегрированные среды разработки (IDE):** Visual Studio и другие IDE часто генерируют GUID автоматически при создании новых COM-классов или интерфейсов через свои мастера.
  + **Программно:** Функция COM API **CoCreateGuid()** генерирует новый GUID во время выполнения программы.

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.



**CoUninitialize()**

* + **Назначение:** Освободить ресурсы COM-библиотеки, выделенные для *текущего потока*. Должна вызываться один раз для каждого успешного вызова CoInitialize или CoInitializeEx в том же потоке.
  + **Кто вызывает:** Тот же код (клиент или EXE-сервер), который вызывал CoInitialize/CoInitializeEx.

**2. Создание Экземпляра Компонента (Клиент)**

* **CoGetClassObject(REFCLSID rclsid, DWORD dwClsContext, COSERVERINFO \*pServerInfo, REFIID riid, void\*\* ppv)**
  + **Назначение:** **Основной способ получить фабрику классов** для заданного компонента. Это первый шаг "двухэтапного" создания объекта.
* **CoCreateInstance(REFCLSID rclsid, LPUNKNOWN pUnkOuter, DWORD dwClsContext, REFIID riid, void\*\* ppv)**
  + **Назначение:** **Наиболее распространенный и удобный способ создать один экземпляр** COM-компонента. Это "одноэтапный" метод для клиента.
  + **Что делает внутри:** Является **оберткой** над последовательностью:
    1. Вызов CoGetClassObject для получения IClassFactory\*.
    2. Вызов IClassFactory::CreateInstance для создания объекта и получения нужного интерфейса (riid).
    3. Вызов IClassFactory::Release для освобождения фабрики.
  + **Параметры (ключевые):**
    1. rclsid: CLSID компонента, который нужно создать.
    2. pUnkOuter: Для агрегации (обычно NULL).
    3. dwClsContext: Тип сервера (как в CoGetClassObject).
    4. riid: IID интерфейса, который клиент хочет получить **от созданного объекта** (например, IID\_IMath).
    5. ppv: Указатель на запрошенный интерфейс созданного объекта.
* **CoCreateInstanceEx(REFCLSID rclsid, ..., COSERVERINFO \*pServerInfo, ULONG cmq, MULTI\_QI \*pResults)**
  + **Назначение:** Расширенная версия CoCreateInstance. Основные отличия:
    1. Явно поддерживает создание объекта на **удаленной машине** через параметр pServerInfo.
    2. Позволяет запросить **несколько интерфейсов** у созданного объекта за один вызов (оптимизация) с помощью массива структур MULTI\_QI.

1. **Предоставление Фабрик Классов (Сервер)**

Мы пишем COM-компонент как DLL (кроме этого мы могли бы сделать его как exe). Со стороны разработчика мы ДОЛЖНЫ прописать DllGetClassObject для того, чтобы СИСТЕМА могла получить фабрику компонентов.  
Потом когда клиент будет использовать CoGetClassObject, она фактически будет получать его через DllGetClassObject. То есть DllGetClassObject - для системы, CoGetClassObject - для клиента. Клиент бы не смог получить фабрику через CoGetClassObject, если бы не DllGetClassObject

* **Общая цель:** И DllGetClassObject (для DLL), и механизм с CoRegisterClassObject (для EXE) служат для того, чтобы **сделать фабрику классов сервера доступной** для COM-подсистемы и, следовательно, для клиентов.
* **DllGetClassObject (для DLL):**
  + **Роль:** **Пассивный ответчик.** DLL ждет, пока COM-система не придет к ней (после загрузки DLL) и не *запросит* фабрику через эту конкретную функцию.
  + **Кто вызывает:** COM-система.
  + **Когда:** В момент, когда фабрика реально нужна (например, во время вызова CoGetClassObject клиентом для этой DLL).
* **CoRegisterClassObject (для EXE):**
  + **Роль:** **Активный регистратор.** EXE-сервер при запуске *сам* создает свои фабрики и *активно сообщает* COM-системе: "Вот моя фабрика для такого-то CLSID, зарегистрируй её, пожалуйста".
  + **Кто вызывает:** Код внутри EXE-сервера.
  + **Когда:** Во время инициализации EXE-сервера, *до* того, как клиенты начнут запрашивать объекты.
* **DllGetClassObject(REFCLSID rclsid, REFIID riid, void\*\* ppv)**
  + **Назначение:** **Точка входа для COM в DLL-сервер**. COM вызывает эту функцию, чтобы получить фабрику классов от DLL.
  + **Кто вызывает:** **COM-библиотека** (в ответ на вызов клиентом CoGetClassObject для этой DLL).
  + **Кто реализует:** **Разработчик DLL-сервера**. Эта функция должна быть экспортируема из DLL.
  + **Что делает (реализация разработчика):**
    1. Проверяет, совпадает ли rclsid с CLSID компонента(ов), поддерживаемых этой DLL.
    2. Создает экземпляр соответствующей фабрики классов (new MathClassFactory).
    3. Вызывает QueryInterface у фабрики, чтобы получить запрошенный riid (обычно IID\_IClassFactory).
    4. Возвращает указатель на интерфейс фабрики через ppv.
* **CoRegisterClassObject(REFCLSID rclsid, LPUNKNOWN pUnk, DWORD dwClsContext, DWORD flags, LPDWORD lpdwRegister)**
  + **Назначение:** **Сообщить системе COM, что фабрика классов готова** принимать запросы на создание объектов. Используется **только EXE-серверами**.
  + **Кто вызывает:** **EXE-сервер** (обычно при старте).
  + **Кто реализует:** COM-библиотека.
  + **Что делает:** Регистрирует активную фабрику классов (pUnk) в глобальной таблице объектов COM для данного CLSID (rclsid) и контекста (dwClsContext). Теперь COM сможет найти эту фабрику, когда клиент вызовет CoGetClassObject.
  + **flags:** Определяют режим использования фабрики (один клиент, много клиентов и т.д.).

**4. Управление Жизненным Циклом DLL (Сервер)**

* **DllCanUnloadNow()**
  + **Назначение:** Позволить COM определить, можно ли безопасно выгрузить DLL из памяти.
  + **Кто вызывает:** **COM-библиотека** (периодически).
  + **Кто реализует:** **Разработчик DLL-сервера**. Эта функция должна быть экспортируема из DLL.
  + **Что делает (реализация разработчика):** Проверяет глобальный счетчик активных объектов и счетчик блокировок (LockServer). Если оба равны нулю, возвращает S\_OK (можно выгружать), иначе S\_FALSE.

**5. Утилиты для GUID (Клиент и Сервер)**

* **CoCreateGuid(GUID\* pguid)**
  + **Назначение:** Сгенерировать новый, уникальный GUID.
  + **Кто вызывает:** Может вызываться кем угодно (клиентом, сервером, утилитами).
  + **Кто реализует:** COM-библиотека.

**Итог по параметрам CreateInstance:**

Как мы видели, IClassFactory::CreateInstance(pUnkOuter, riid, ppv):

* pUnkOuter: Обычно NULL. Используется для агрегации.
* riid: Какой интерфейс вы хотите получить от **нового объекта**.
* ppv: Куда поместить указатель на этот интерфейс.

# Про суррогат

**Проблема:**

* У вас есть COM-компонент, упакованный в **DLL** (In-Process сервер).
* DLL-серверы обычно работают **внутри процесса клиента**. Это быстро, но имеет недостатки:
  + Нельзя вызвать из другого компьютера (нет DCOM).
  + Если DLL-сервер "упадет", он "утащит" за собой весь клиентский процесс.
  + Клиент и DLL работают под одним пользователем, что не всегда хорошо для безопасности.
* Чтобы решить эти проблемы, нужен **EXE-сервер** (Out-of-Process), который работает в отдельном процессе.
* Но переписывать рабочую DLL в полноценный EXE-сервер — это дополнительная работа (нужно создать EXE-проект, написать точку входа, обработку сообщений, зарегистрировать фабрики и т.д.).

Windows предоставляет готовую **стандартную программу-обертку** — **dllhost.exe**.

Эта программа — **пустой контейнер** или "хост". Сама по себе она не делает ничего полезного, кроме как умеет **загружать внутрь себя COM DLL-серверы**.

**Как это работает:**

Вы **специальным образом регистрируете** вашу DLL в реестре Windows, указывая, что она должна запускаться через суррогат (DllSurrogate в ключе AppID).

Когда клиент запрашивает создание вашего компонента как **Out-of-Process** (CLSCTX\_LOCAL\_SERVER или CLSCTX\_REMOTE\_SERVER), COM видит эту специальную регистрацию.

Вместо того чтобы загрузить DLL в клиентский процесс (как для CLSCTX\_INPROC\_SERVER), COM **запускает экземпляр dllhost.exe**.

COM говорит этому dllhost.exe: "Загрузи, пожалуйста, вот эту DLL (вашу DLL)".

dllhost.exe загружает вашу DLL в свое собственное адресное пространство.

Теперь для клиента ваш компонент выглядит и ведет себя как **обычный EXE-сервер**: он работает в отдельном процессе (dllhost.exe), и взаимодействие с ним идет через механизм прокси/заглушек и маршалинг.

# Структура проекта

1. **ComSampleClient**: Это клиентское приложение (.exe). Его задача — создавать экземпляры COM-объектов, предоставляемых серверами, и вызывать их методы. Он демонстрирует, как использовать COM-объекты независимо от того, где они реализованы (в DLL в том же процессе, в DLL в другом процессе через суррогат или в службе).
2. **ComSampleProxy**: Это DLL-библиотека, содержащая *прокси* и *стаб* (proxy/stub) для интерфейса IComTest. Она генерируется автоматически компилятором MIDL на основе IDL-определения интерфейса. Эта DLL необходима для *маршалинга* — упаковки и распаковки данных и вызовов методов при взаимодействии между разными процессами (клиент <-> внепроцессный сервер/служба) или разными апартаментами (потоковыми моделями) внутри одного процесса.
3. **ComSampleServer**: Это COM-сервер, реализованный в виде DLL (.dll). Он реализует COM-объект CComServerTest, который предоставляет интерфейс IComTest. Эта DLL может работать как:
   * **Внутрипроцессный сервер (In-Process Server, InProc)**: Загружается непосредственно в адресное пространство клиента.
   * **Внепроцессный сервер (Out-of-Process Server, Local Server)**: Запускается в отдельном процессе. В данном случае используется стандартный суррогатный хост Windows — DllHost.exe, который загружает эту DLL. Настроен так через регистрацию AppID).
4. **ComSampleService**: Это COM-сервер, реализованный в виде исполняемого файла (.exe), который работает как Windows-служба. Он также реализует COM-объект (CComServiceTest), предоставляющий интерфейс IComTest, но его жизненный цикл управляется Менеджером Управления Службами (Service Control Manager, SCM) Windows. Он всегда является внепроцессным сервером.

Основная цель проекта: Показать, как один и тот же интерфейс (IComTest) может быть реализован разными способами (DLL, Service) и как клиент может взаимодействовать с этими реализациями, используя стандартные механизмы COM, абстрагируясь от деталей реализации и местоположения сервера. Метод WhoAmI в интерфейсе как раз и служит для того, чтобы сервер сообщил клиенту, в каком процессе он запущен.

# ComSampleProxy

1. **IDL (Interface Definition Language / Язык Описания Интерфейсов):**
   * Это **язык описания**, похожий на C/C++, но не являющийся языком программирования.
   * IDL — это общий термин для любого языка, предназначенного для описания интерфейсов программного обеспечения. Основная цель — определить "контракт" между компонентами.
   * Является **независимым от языка программирования**. Определения в IDL могут быть использованы для создания кода на разных языках (C++, C#, VB и т.д.).
   * **Результат:** IDL-файл (.idl) - это исходный код для описания контракта компонента.
2. **MIDL (Microsoft Interface Definition Language Compiler / Компилятор Языка Описания Интерфейсов Microsoft):**
   * **Назначение:** Обрабатывать файлы .idl (написанные на языке IDL).
   * **Результат:** На основе IDL-файла компилятор MIDL генерирует:
     + **Библиотеку типов (.tlb):** Машиночитаемое описание интерфейсов и типов для использования другими приложениями и средами разработки.
     + **Заголовочные файлы C/C++ (.h):** Определения интерфейсов и типов для использования в C/C++ коде.
     + **Код прокси/заглушек (proxy/stub):** C-код, необходимый для передачи вызовов методов и данных между различными процессами или потоковыми моделями (маршалинг).

**IComTest.idl**

Это **описание интерфейса**, не зависящее от языка программирования. Это контракт, который должны соблюдать и сервер (реализуя интерфейс), и клиент (используя интерфейс).

Квадратные скобки [] перед определением интерфейса (или другого элемента IDL) используются для указания **атрибутов**. Атрибуты предоставляют метаданные об элементе, который они описывают. Эти метаданные используются компилятором MIDL (Microsoft Interface Definition Language) и средой выполнения COM.

1. **object**:
   * Указывает, что IComTest является **COM-интерфейсом**, а не интерфейсом DCE RPC (более старая технология, на которой основан COM).
   * Для COM-интерфейсов это стандартный атрибут. Он подразумевает, что интерфейс должен (прямо или косвенно) наследоваться от IUnknown. Хотя в вашем примере наследование от IUnknown указано явно (: IUnknown), атрибут object все равно важен для MIDL, чтобы правильно обработать интерфейс как COM-интерфейс.
2. **uuid(db24e16f-aa47-4fae-83bb-9f18ebd1e9bc)**:
   * Это **самый важный** атрибут для интерфейса.
   * uuid означает Universally Unique Identifier (Универсальный Уникальный Идентификатор), также известный как GUID (Globally Unique Identifier - Глобальный Уникальный Идентификатор).
   * Значение в скобках (db24e16f-...) - это **идентификатор интерфейса (Interface ID, IID)** для IComTest.
   * COM использует этот уникальный номер для идентификации интерфейса во время выполнения. Когда вы вызываете QueryInterface, вы передаете IID интерфейса, который хотите получить. Этот IID должен быть уникальным во всем мире, чтобы избежать конфликтов между разными интерфейсами от разных разработчиков.
   * Эти UUID/GUID генерируются специальными утилитами (например, uuidgen.exe или средствами Visual Studio).
3. **version(1.0)**:
   * Указывает версию интерфейса. Обычно используется формат major.minor.
   * Эта информация записывается в библиотеку типов (.tlb), которая генерируется из IDL-файла.
4. **helpstring("IComTest interface")**:
   * Предоставляет краткое, понятное человеку описание интерфейса.
   * Эта строка также записывается в библиотеку типов.

**IUnknown**: Это базовый интерфейс, от которого **должны** (прямо или косвенно) наследоваться *все* COM-интерфейсы. IUnknown предоставляет фундаментальные механизмы COM.

Файл IComTest\_h.h, IComTest\_i.c, IComTest\_p.c, dlldata.c **генерируется автоматически** компилятором **MIDL** при обработке файла IComTest.idl.

* **Файл: IComTest\_h.h (Заголовочный файл C/C++)**
  + **Назначение:** Содержит объявление интерфейса IComTest в виде, понятном компилятору C++. Это как бы "перевод" IDL-определения на C++. Его подключают (#include) и клиент, и сервер.
  + **Ключевое содержимое:**
    - Объявление структуры IComTestVtbl: Это описание **виртуальной таблицы методов (VTable)** для IComTest. Она включает указатели на функции QueryInterface, AddRef, Release (унаследованные от IUnknown) и на нашу функцию WhoAmI.
    - Объявление самого интерфейса как структуры, содержащей указатель на VTable (struct IComTestVtbl \*lpVtbl;).
    - Объявление **константы IID\_IComTest** типа IID (который является псевдонимом для GUID).
    - Макросы для удобного вызова методов через указатель на интерфейс (например, IComTest\_WhoAmI(pComTest, ...)).
* **Файл: IComTest\_i.c (Определение GUID)**
  + **Назначение:** Содержит **определение** (выделение памяти и инициализацию) константы IID\_IComTest.
  + **COM Концепция (Инициализация GUID):** Чтобы избежать ошибок линковки "символ определен несколько раз", GUID'ы должны быть *объявлены* (через DEFINE\_GUID в .h файле) во многих местах, но *определены* (память выделена) только в **одном** .c или .cpp файле во всем проекте (DLL или EXE). Это достигается включением #include <initguid.h> **перед** включением заголовочника с DEFINE\_GUID (например, IComTest\_h.h) в том .c/.cpp файле, где нужно определение. В вашем клиенте (Main.cpp) как раз используется <initguid.h>. В данном случае, #include <initguid.h> в main.cpp используется для того, чтобы **заставить определения CLSID\_CComServerTest и CLSID\_CComServiceTest (из ComSampleServerGuids.h и ComSampleServiceGuids.h) быть созданными именно в компилируемом коде клиента**.

С initguid.h – определение, без – объявление.

* **IComTest\_p.c (Generated Proxy/Stub Code):**
  + Он содержит код **прокси** и **стаба** для интерфейса IComTest.
  + **Прокси:** Реализует интерфейс IComTest на стороне клиента. Когда клиент вызывает метод (например, WhoAmI), код прокси:
    1. Собирает параметры вызова.
    2. Упаковывает (маршалит) их в стандартный формат (NDR - Network Data Representation).
    3. Передает упакованные данные через механизм RPC/COM в процесс сервера.
  + **Стаб:** Находится на стороне сервера. Он:
    1. Принимает упакованные данные от прокси.
    2. Распаковывает (демаршалит) параметры.
    3. Вызывает реальный метод WhoAmI у объекта CComServerTest или CComServiceTest.
    4. Получает результат и выходные параметры от реального метода.
    5. Упаковывает результат и выходные параметры.
    6. Отправляет их обратно прокси.
  + Прокси затем распаковывает результат и возвращает его клиенту.
  + Весь этот процесс **прозрачен** для клиента и сервера. Клиент думает, что вызывает метод напрямую, сервер думает, что его вызывают напрямую.
  + Содержит таблицы форматов данных (IComTest\_\_MIDL\_TypeFormatString, IComTest\_\_MIDL\_ProcFormatString), V-таблицы для прокси и стаба (\_IComTestProxyVtbl, \_IComTestStubVtbl), и функции для маршалинга/демаршалинга конкретных типов данных.

**dlldata.c (Generated DLL Data):**

* + Вспомогательный файл для сборки **Proxy/Stub DLL**.

**IComTest.def (Module Definition File):**

* + Определяет имя выходной DLL (ComServerProxy.dll).

# Структура клиента

initguid.h: Используется *один раз* в проекте (обычно в одном .cpp файле), чтобы определить фактические значения GUID, а не только их объявления (extern).

#include <combaseapi.h> // Основные определения COM API (CoInitializeEx, CoCreateInstance, ...)

#include <initguid.h> // !! Важно: Переключает DEFINE\_GUID в режим ОПРЕДЕЛЕНИЯ GUID'ов в этом файле

IComTest\_h.h: Заголовочный файл, сгенерированный MIDL из IComTest.idl. Содержит объявление интерфейса IComTest и его IID (IID\_IComTest).

ComSampleServerGuids.h: Содержит определение CLSID (CLSID\_CComServerTest) для класса в ComSampleServer.

ComSampleServiceGuids.h: Содержит определение CLSID (CLSID\_CComServiceTest) для класса в ComSampleService.

Принимает rclsid (какой класс создать), dwCoInit (в каком апартаменте инициализировать поток), dwClsContext (какой тип сервера искать).

HRESULT hr = CoInitializeEx(NULL, dwCoInit);: Инициализирует COM для текущего потока в указанном апартаменте (STA или MTA). **Каждый поток, использующий COM, должен вызвать это.**

if (SUCCEEDED(hr)): Проверка на успешность инициализации. SUCCEEDED/FAILED – макросы для проверки HRESULT.

IComTest \*pComTest;: Объявление указателя на интерфейс.

hr = CoCreateInstance(...): Пытается создать экземпляр COM-объекта.

CoTaskMemAlloc, CoTaskMemRealloc и CoTaskMemFree составляют **стандартный механизм управления памятью** в COM. Это гарантирует, что память, выделенная одним модулем (например, COM-сервером), может быть безопасно освобождена другим модулем (например, клиентом), даже если они написаны на разных языках или используют разные среды выполнения.

**IID\_PPV\_ARGS** — это **макрос-помощник**. Он нужен, чтобы **проще и безопаснее вызывать** функции вроде QueryInterface или CoCreateInstance. Меньше шансов ошибиться с IID.

# Структура SampleServer

**ComSampleServerGuids.h:**

* DEFINE\_GUID(CLSID\_CComServerTest, ...): Определяет CLSID для класса CComServerTest. Этот GUID уникально идентифицирует *реализацию*.

**ComSampleServerCreateInstances.h:**

* HRESULT CComServerTest\_CreateInstance(...): Объявление функции-создателя для класса CComServerTest. Эта функция будет использоваться фабрикой классов.

**CComServerTest.cpp:**

* **Класс CComServerTest:**
  + class CComServerTest : public IComTest: Класс наследует от IComTest. Поскольку IComTest наследует от IUnknown, CComServerTest неявно наследует и IUnknown.
* **QITAB** — это структура для описания пар "Класс/Интерфейс". QITABENT — макрос для удобного заполнения этой структуры.
* **QITABENT(CComServiceTest, IComTest):** Этот макрос генерирует запись в таблице, которая говорит примерно следующее: "Если запрошен интерфейс IComTest (его IID берется из \_\_uuidof(IComTest)), то для объекта, чей *конкретный тип* - CComServiceTest, нужно вернуть указатель, полученный путем приведения типа к IComTest\*".

**QISearch** — это вспомогательная функция (из библиотеки shlwapi), которая автоматизирует поиск по таблице интерфейсов. Она проверяет riid по записям в qit. Если находит совпадение (или если запрашивается IUnknown, она обычно берет первый интерфейс в таблице и запрашивает у него IUnknown), она выполняет приведение типа (static\_cast), вызывает AddRef() и помещает результат в ppv.

**Реализация IComTest:**

* **CComServerTest\_CreateInstance Function:**
  + Фабричная *функция* (не путать с фабричным *классом*).
* **ClassFactory.h:**
  + typedef HRESULT (\*PFNCREATEINSTANCE)(...): Определяет тип указателя на функцию-создатель экземпляра объекта (как CComServerTest\_CreateInstance).
  + HRESULT CClassFactory\_CreateInstance(...): Объявление функции, которая создает *объект фабрики классов*.
* **ClassFactory.cpp:**
  + **Класс CClassFactory:**
    - Реализует IClassFactory.
    - Содержит \_cRef (счетчик ссылок для самого объекта фабрики) и \_pfnCreateInstance (указатель на функцию, которая создает реальные COM-объекты, например CComServerTest\_CreateInstance).
    - **IUnknown:** Стандартная реализация AddRef, Release, QueryInterface (использует QITAB и QISearch).
    - **IClassFactory:**
      * LockServer(\_\_in BOOL fLock): Увеличивает/уменьшает глобальный счетчик блокировок g\_cServerLocks. Это позволяет клиенту удерживать DLL в памяти.
      * CreateInstance(...):
        + Этот CClassFactory - универсальная (generic) фабрика. Она не "зашита" на создание только CComServiceTest. Она спроектирована так, чтобы мочь создавать любой COM-объект, если ей передать правильную "инструкцию" по его созданию.
        + PFNCREATEINSTANCE \_pfnCreateInstance; Это член класса, который является указателем на функцию. PFNCREATEINSTANCE - это псевдоним (typedef) для типа функции, которая принимает REFIID riid и void \*\*ppv и возвращает HRESULT.
        + Когда создается экземпляр CClassFactory, ему в конструктор передается указатель на функцию-создатель (например, адрес CComServiceTest\_CreateInstance) и сохраняется в \_pfnCreateInstance.
  + **CClassFactory\_CreateInstance Function:**
    - Создает экземпляр *самой фабрики* (new CClassFactory(pfnCreateInstance))

• Dll.h: Объявляет глобальные переменные g\_cRefDll и g\_cServerLocks.

**Dll.cpp:**

* **AppID (Application ID)** – уникальный идентификатор, который группирует один или несколько COM-классов (CLSID), предназначенных для запуска в *одном серверном процессе*.
* Здесь он используется, чтобы указать COM, что классы, связанные с этим AppID, должны запускаться не внутри процесса клиента (In-Process), а в отдельном процессе.
* struct CLASS\_INFO: Определяет структуру для хранения всей необходимой информации о *каждом* COM-классе, который реализует эта DLL.
  + rclsid: Уникальный идентификатор класса (CLSID).
  + pwszName: Человекочитаемое имя класса (используется для записи в реестр).
  + pwszThreadModel: Модель потоков (Threading Model) – очень важна для COM! Указывает, как объекты этого класса взаимодействуют с потоками. Возможные значения:
    - Apartment (или Single): Объект может быть доступен только из одного потока (его "квартиры"). COM обеспечивает сериализацию вызовов из других потоков.
    - Free: Объект может быть доступен из нескольких потоков одновременно. Объект *сам* должен обеспечивать потокобезопасность.
    - Both: Объект потокобезопасен и может работать как в Apartment, так и в Free-threaded окружении.
    - Neutral: Для Windows Runtime, не типично для классического COM.
  + pwszAppId: Указатель на строку AppID. Если класс должен быть доступен как Out-of-Process через суррогат, здесь указывается APPID\_LOCALSERVER. Если пустая строка (L""), класс регистрируется только как In-Process.
  + pfnCreateInstance: **Указатель на функцию**, которая *фактически создает* экземпляр этого конкретного класса (например, CComServerTest\_CreateInstance). Это связующее звено с универсальной фабрикой CClassFactory.

g\_Classes[]: Глобальный **массив** структур CLASS\_INFO. Это **центральная таблица** для DLL.

g\_hInstance: Глобальная переменная для хранения хендла (описателя) экземпляра загруженной DLL.

1. **DllGetClassObject (Стандартная экспортируемая функция):**
   * **Назначение:** Это **основная точка входа** для COM, когда клиент запрашивает создание объекта (через CoCreateInstance или CoGetClassObject). COM вызывает эту функцию, передавая CLSID нужного объекта и IID интерфейса, который нужен от *фабрики классов* (обычно IID\_IClassFactory).
   * **Реализация:**
     1. Проверяет валидность выходного параметра ppv.
     2. **Итерирует по глобальной таблице g\_Classes**.
     3. Сравнивает запрошенный rclsid с rclsid каждой записи в таблице.
     4. Если найдено совпадение:
        + Вызывает **CClassFactory\_CreateInstance** (функцию, которая создает экземпляр *нашей универсальной фабрики* CClassFactory).
        + Передает ей **указатель на функцию создания *конкретного* объекта** (g\_Classes[i].pfnCreateInstance), взятый из найденной записи таблицы.
        + Передает riid (обычно IID\_IClassFactory) и ppv, чтобы CClassFactory\_CreateInstance вернула указатель на интерфейс фабрики.
        + Устанавливает флаг fFound и выходит из цикла.
     5. Если цикл завершился без совпадения, возвращает CLASS\_E\_CLASSNOTAVAILABLE.
     6. Возвращает HRESULT.

**DllCanUnloadNow (Стандартная экспортируемая функция):**

* + **Назначение:** Позволяет COM периодически спрашивать у DLL, можно ли ее выгрузить из памяти.

**Функции Регистрации/Разрегистрации:**

* + Эти функции вызываются утилитой regsvr32.exe YourDllName.dll (для регистрации) и regsvr32.exe /u YourDllName.dll (для разрегистрации).
  + **\_WriteRegistryEntries (Вспомогательная):**
    - Низкоуровневая функция для записи *одной* пары "ключ-значение" в реестр.
  + **\_RegisterClass (Вспомогательная):**
    - Регистрирует *один* COM-класс на основе информации из CLASS\_INFO.
    - Циклически вызывает \_WriteRegistryEntries для каждой записи в rgRegEntries.
  + **\_UnregisterClass (Вспомогательная):**
    - Удаляет регистрацию *одного* COM-класса.
  + **\_RegisterAppId (Вспомогательная):**
    - Регистрирует AppID, необходимый для суррогатной активации.
      * (Default Value) = Имя хоста (например, "Com Sample Host").
      * DllSurrogate = **Пустая строка**. Это специальное значение, которое **указывает COM использовать стандартный системный суррогат DllHost.exe**. Если бы здесь был путь к EXE, использовался бы он.
      * RunAs = "Interactive User". Определяет учетную запись, от имени которой будет запущен суррогатный процесс DllHost.exe. "Interactive User" означает, что он будет запущен от имени пользователя, вошедшего в систему в данный момент. (Другие опции включают конкретного пользователя или системные учетные записи, но требуют настройки прав).
    - Циклически вызывает \_WriteRegistryEntries.
  + **\_UnregisterAppId (Вспомогательная):**
    - Удаляет регистрацию AppID.
  + **DllRegisterServer (Стандартная экспортируемая функция):**
    - Вызывается regsvr32.exe.
    - Циклически проходит по g\_Classes и вызывает \_RegisterClass для каждого класса.
    - Вызывает \_RegisterAppId *один раз* для APPID\_LOCALSERVER.
  + **DllUnregisterServer (Стандартная экспортируемая функция):**
    - Вызывается regsvr32.exe /u.
    - Циклически проходит по g\_Classes и вызывает \_UnregisterClass для каждого класса. В отличие от регистрации, она **продолжает работу даже при ошибках**, пытаясь удалить как можно больше записей.
    - Вызывает \_UnregisterAppId *один раз*.

1. **DllMain (Стандартная точка входа DLL):**
   * **Назначение:** Это функция, вызываемая операционной системой при загрузке и выгрузке DLL из процесса, а также при создании/завершении потоков в этом процессе.
   * **Реализация:**
     + Проверяет причину вызова (dwReason).
     + Если причина DLL\_PROCESS\_ATTACH (DLL загружается в процесс):
       - Сохраняет хендл модуля (hModule) в глобальную переменную g\_hInstance.
       - DisableThreadLibraryCalls(g\_hInstance): Это оптимизация. Она говорит ОС не вызывать DllMain для этой DLL при каждом создании/завершении потока (DLL\_THREAD\_ATTACH, DLL\_THREAD\_DETACH). Это делается, если DLL не выполняет никакой специфической для потока инициализации/очистки, что часто бывает в COM DLL.

# Структура SampleService

* **Infrastructure.h:**
  + Определяет константы: SERVICE\_NAME, SERVICE\_DISPLAY\_NAME, SERVICE\_APPID.
  + Объявляет функции, используемые в разных частях проекта службы:
    - ServiceMain: Точка входа службы Windows.
    - RegisterClass, UnregisterClass: Регистрация COM-класса (специфичная для службы).
    - RegisterAppID, UnregisterAppID: Регистрация AppID (специфичная для службы).
    - RegisterService, UnregisterService: Регистрация самой службы в SCM.
    - StartFactories, StopFactories: Запуск/остановка регистрации фабрик классов COM.

**Отличие класса:** В конструкторе и деструкторе **нет** манипуляций с g\_cRefDll или g\_cServerLocks. Жизненный цикл EXE-сервера (особенно службы) управляется иначе – через SCM и регистрацию/отзыв фабрик классов, а не счетчиками ссылок на DLL.

* g\_cRefDll (глобальный счетчик ссылок на DLL) нужен для реализации DllCanUnloadNow.
* EXE-сервер работает в **отдельном процессе**. У него нет функции DllCanUnloadNow.
* COM **не спрашивает** EXE-процесс, можно ли его завершить, основываясь на счетчиках объектов внутри него таким же образом, как у DLL.
* **Как COM решает, когда завершить EXE-сервер?** COM следит за тем, есть ли у него **активные зарегистрированные фабрики классов** (CoRegisterClassObject) или **существующие COM-соединения** к объектам внутри этого EXE.
* Когда EXE-сервер (служба) запускается, он вызывает CoRegisterClassObject для каждой фабрики, которую он предоставляет. Это говорит COM: "Я жив и готов создавать объекты вот этих CLSID".
* Когда последний клиент отключается от всех объектов сервера *И* сервер отзывает все свои фабрики классов (CoRevokeClassObject), COM понимает, что сервер больше не нужен, и может инициировать его завершение (или просто перестать его поддерживать активным, если он не служба).
* Для **службы** управление еще сложнее: SCM (Service Control Manager) также играет роль. Служба может работать, даже если нет активных COM-клиентов, и будет остановлена только по команде SCM (или если сама решит остановиться). COM-активация просто *запускает* службу, если она не запущена.

**Отличие фабрики:** Метод LockServer **ничего не делает** и просто возвращает S\_OK. Для EXE-сервера блокировка не имеет смысла в том же виде, что для DLL; COM и SCM управляют временем жизни процесса.

**Register.cpp:** Содержит функции регистрации, **специфичные для EXE-службы**.

* **RegisterClass(...):**
  + Создает ключ HKEY\_CLASSES\_ROOT\CLSID\{CLSID}.
  + Устанавливает значение по умолчанию (имя класса).
  + Устанавливает значение AppID = pszAppID (связывает CLSID с AppID службы).
  + **Отличие от DLL:** Не создает подраздел InprocServer32. Вместо этого COM будет использовать AppID для поиска информации о запуске сервера.
* **UnregisterClass(...):** Удаляет ключ CLSID\{CLSID}.
* **RegisterAppID(...):**
  + Создает ключ HKEY\_CLASSES\_ROOT\AppID\{APPID}.
  + Устанавливает значение по умолчанию (имя приложения).
  + Устанавливает значение LocalService = pszServiceName (имя службы Windows). **Ключевой момент!** Это значение говорит COM, что для активации сервера с этим AppID нужно обратиться к SCM и запустить службу с указанным именем.
  + **Отличие от DLL:** Не использует DllSurrogate или RunAs (хотя RunAs можно было бы указать, если бы служба запускалась не от LocalSystem).
* **UnregisterAppID(...):** Удаляет ключ AppID\{APPID}.
* **RegisterService(...):**
  + Использует API SCM (OpenSCManager, CreateService) для регистрации EXE как службы Windows.
  + GetModuleFileName(nullptr, ...): Получает путь к текущему EXE.
  + CreateService: Указывает имя службы, отображаемое имя, тип (SERVICE\_WIN32\_OWN\_PROCESS), тип запуска (SERVICE\_DEMAND\_START - запускать по требованию), путь к EXE. nullptr для имени пользователя означает запуск от LocalSystem.
  + Возвращает S\_FALSE, если служба уже существует.
* **UnregisterService(...):**
  + Использует API SCM (OpenSCManager, OpenService, DeleteService) для удаления регистрации службы.

**Service.cpp: Содержит код для реализации логики службы Windows.**

* Класс CMpcService: Инкапсулирует управление состоянием службы.
  + \_csLock: Критическая секция для синхронизации доступа к состоянию службы.
  + \_hServiceStatus: Хэндл статуса службы, полученный от SCM.
  + \_serviceStatus: Структура, хранящая текущий статус службы.
  + \_hServiceStoppingEvent: Событие, которое сигнализирует основному потоку службы о необходимости завершения работы.
* g\_pSvc: Глобальный указатель на экземпляр CMpcService.
* ServiceMain (глобальная функция, не метод):
  + Точка входа службы. Вызывается SCM при запуске службы.
  + Создает экземпляр CMpcService.
  + Вызывает g\_pSvc->ServiceMain().
  + Удаляет экземпляр по завершении.
* CMpcService::s\_ServiceHandler (static):
  + Обработчик управляющих кодов от SCM (например, STOP, PAUSE). Регистрируется с помощью RegisterServiceCtrlHandlerEx.
  + Вызывается SCM в отдельном потоке.
  + Передает вызов реальному обработчику в g\_pSvc.
* CMpcService::ServiceHandler(...):
  + Реальный обработчик кодов.
  + При получении SERVICE\_CONTROL\_STOP:
    - Устанавливает статус SERVICE\_STOP\_PENDING с помощью \_SetStatus.
    - Устанавливает событие \_hServiceStoppingEvent, чтобы сигнализировать основному потоку.
* CMpcService::\_SetStatus(...):
  + Обновляет структуру \_serviceStatus.
  + Вызывает SetServiceStatus, чтобы уведомить SCM об изменении состояния.
* CMpcService::\_StartService():
  + Создает событие \_hServiceStoppingEvent.
  + Регистрирует обработчик s\_ServiceHandler с помощью RegisterServiceCtrlHandlerEx, получает \_hServiceStatus.
  + Устанавливает статус SERVICE\_START\_PENDING.
  + StartFactories(): Вызывает функцию из Main.cpp для регистрации фабрик классов COM (CoRegisterClassObject).
  + Устанавливает статус SERVICE\_RUNNING.
* CMpcService::\_StopService():
  + StopFactories(): Вызывает функцию из Main.cpp для отмены регистрации фабрик (CoRevokeClassObject).
  + Устанавливает статус SERVICE\_STOPPED.
  + Закрывает хэндл события \_hServiceStoppingEvent.
* CMpcService::ServiceMain (метод):
  + Основной поток службы.
  + CoInitializeEx(nullptr, COINIT\_MULTITHREADED): Инициализирует COM в MTA для основного потока службы. Это обычная практика для служб.
  + CoInitializeSecurity(...): Очень важно для Out-of-Proc серверов. Настраивает безопасность COM для процесса. Без этого вызовы из других процессов могут блокироваться. Параметры здесь довольно стандартные (разрешают подключение, используют аутентификацию NTLM/Kerberos, разрешают имперсонацию). EOAC\_NONE отключает некоторые доп. возможности.
  + \_StartService(): Запускает службу и регистрирует фабрики COM.
  + WaitForSingleObject(\_hServiceStoppingEvent, INFINITE): Блокирует основной поток до тех пор, пока не будет получен сигнал остановки (событие установлено в ServiceHandler).
  + \_StopService(): Останавливает службу и отменяет регистрацию фабрик.
  + CoUninitialize(): Деинициализирует COM.
* **Main.cpp:**
  + DLL-сервер **не создает** фабрики заранее при загрузке.
  + Вместо этого COM вызывает экспортируемую функцию DllGetClassObject, когда ей нужна фабрика для конкретного CLSID.
  + Внутри DllGetClassObject вызывается CClassFactory\_CreateInstance, которая создает **новый** экземпляр фабрики *каждый раз* (или может использовать какой-то пул, но в данном примере создает новый).
  + DllGetClassObject возвращает указатель на эту свежесозданную фабрику COM.
  + COM использует эту фабрику (вызывает CreateInstance у нее) и затем **сама вызывает Release** на указателе фабрики, который она получила от DllGetClassObject.
  + DLL не нужно хранить указатели на фабрики, потому что она отдает их COM по запросу, и COM сама управляет их временем жизни (через AddRef/Release).
* **StartFactories():**
  + Вызывает CClassFactory\_CreateInstance, чтобы создать объект фабрики для класса.
  + CoRegisterClassObject(...): **Регистрирует фабрику классов в COM.** Сообщает COM, что этот процесс теперь может создавать объекты данного CLSID (g\_Classes[i].rclsid).
  + CLSCTX\_LOCAL\_SERVER: Указывает, что это фабрика для локального сервера.
  + REGCLS\_MULTIPLEUSE: Означает, что одна фабрика может использоваться для создания множества объектов.
  + &g\_Classes[i].dwRegister: Получает cookie для отмены регистрации.
* **StopFactories():**
  + Если dwRegister не 0, вызывает CoRevokeClassObject(g\_Classes[i].dwRegister), чтобы отменить регистрацию фабрики.
* **ExeRegisterServer() / ExeUnregisterServer():**
  + Функции верхнего уровня для регистрации/отмены регистрации **всего** приложения-службы.
  + Вызывают соответствующие функции из Register.cpp: RegisterService/UnregisterService, RegisterAppID/UnregisterAppID, RegisterClass/UnregisterClass (для всех классов в g\_Classes).
  + **wmain(...):**
    - **Точка входа EXE.**
    - Анализирует аргументы командной строки:
      * /RegisterServer: Вызывает ExeRegisterServer().
      * /UnregisterServer: Вызывает ExeUnregisterServer().
      * -Embedding: **Специальный аргумент.** Передается COM/SCM, когда они запускают этот EXE для активации COM-объекта. В этом случае просто устанавливаем fStartService = true.
      * Без аргументов или неизвестные аргументы: По умолчанию fStartService = true (предполагается запуск как службы).
    - if (fStartService): Если нужно запустить службу:
      * Создается SERVICE\_TABLE\_ENTRY, связывающая имя службы (SERVICE\_NAME) с функцией точки входа службы (ServiceMain из Service.cpp).
      * StartServiceCtrlDispatcher(...): Передает управление SCM. SCM запустит ServiceMain в новом потоке. Эта функция блокируется до завершения работы службы.
  + Возвращает 0 при успехе или код ошибки HRESULT.
* **Счётчик ссылок объекта (IUnknown): Определяет время жизни одного экземпляра объекта. Управляется клиентами через AddRef/Release.**
* **Счётчик активных объектов/блокировок сервера: Определяет время жизни всего сервера (DLL/EXE). Управляется конструкторами/деструкторами объектов и вызовами LockServer.**
* **Все методы COM-интерфейсов обязаны использовать соглашение о вызове \_\_stdcall (или его эквивалент в других языках).**
* Почти все методы COM-интерфейсов **должны** возвращать значение типа HRESULT. Исключение: **IUnknown::AddRef() и IUnknown::Release()**

**stdcall — это соглашение о вызове функций (набор правил), где:**

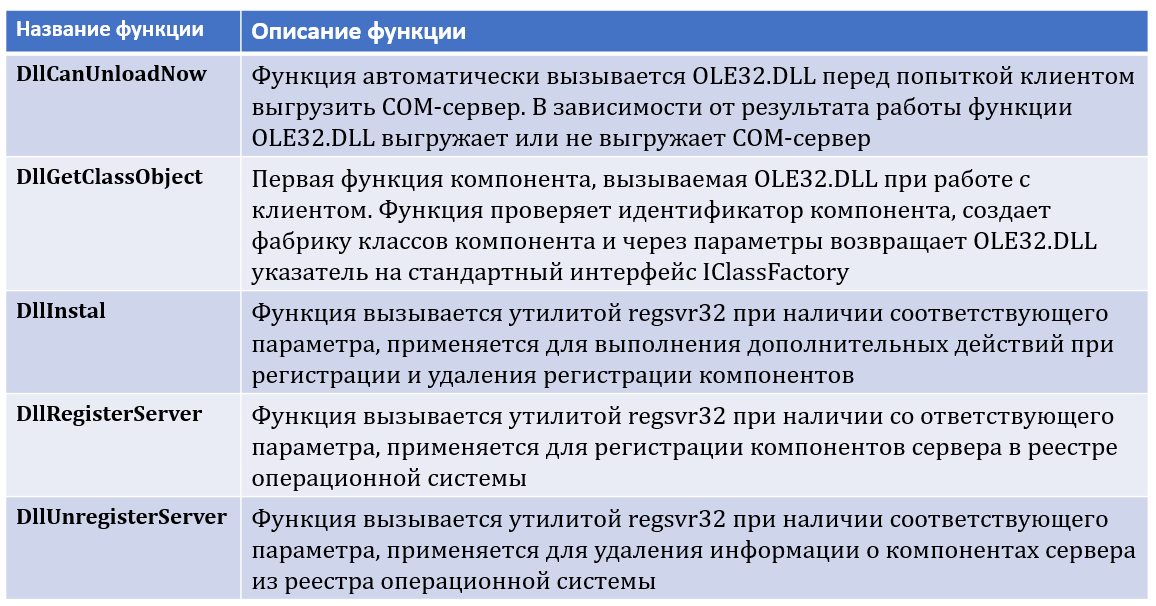
1. **Вызываемая функция (callee) отвечает за очистку стека от аргументов перед возвратом.**
2. **Аргументы передаются в стек справа налево.**

**Вкратце, клиент должен знать:**

* **ЧТО создать (CLSID).**
* **КАК с этим общаться (хотя бы один IID и определение методов этого интерфейса).**

Основная системная библиотека, отвечающая за **фундаментальную COM-инфраструктуру** в Windows, это:

* **ole32.dll**

****

* **Термин "контейнер" больше подчеркивает *физическую* сторону — это файл (DLL/EXE), который "содержит" в себе код компонентов.**
* **Термин "сервер" больше подчеркивает *функциональную* роль — этот файл "обслуживает" клиентов, предоставляя им доступ к функциональности компонентов**

 Пространство Процесса (Самое Главное Отличие)

DLL (In-Process - "внутри процесса"):

Компонент загружается и выполняется внутри адресного пространства клиентского процесса.

Клиент и компонент работают в одном и том же процессе.

Использует память и ресурсы клиентского процесса.

EXE (Out-Of-Process - "вне процесса"):

Компонент выполняется в своем собственном, отдельном процессе.

Клиент и сервер — это два разных процесса Windows.

Имеет свое собственное адресное пространство, память и ресурсы.

2. Производительность

DLL (In-Proc)

Значельно быстрее. Вызовы методов компонента похожи на обычные вызовы функций внутри одного процесса (через таблицу виртуальных методов, v-table). Накладные расходы минимальны (в основном проверка потоковой модели апартамента).

EXE (Out-Of-Proc):

Медленнее. Каждый вызов метода требует межпроцессного взаимодействия (IPC - Inter-Process Communication). Это включает:

Маршалинг: Упаковка параметров вызова в некий формат для передачи между процессами (COM делает это автоматически с помощью прокси и стаба).

Переключение контекста: Операционная система переключается между клиентским и серверным процессами.

Демаршалинг: Распаковка параметров в серверном процессе.

То же самое в обратную сторону для возвращаемых значений.

Накладные расходы на каждый вызов существенны.

3. Стабильность и Изоляция

DLL (In-Proc):

Меньшая изоляция. Если DLL-компонент вызывает критическую ошибку (например, падает из-за ошибки доступа к памяти), он уничтожает весь клиентский процесс.

EXE (Out-Of-Proc):

Большая изоляция. Если EXE-сервер падает, клиентский процесс обычно продолжает работать. Клиент просто получит ошибку (например, RPC\_S\_SERVER\_UNAVAILABLE) при следующей попытке вызвать метод упавшего сервера. Сбой сервера изолирован от клиента.

4. Безопасность и Разрешения

DLL (In-Proc):

Выполняется с теми же правами и под той же учетной записью пользователя, что и клиентский процесс. Он не может иметь больше прав, чем у клиента.

EXE (Out-Of-Proc):

Может выполняться под другой учетной записью пользователя или с другими правами доступа, чем у клиента. Это настраивается при регистрации сервера (через AppID) или во время активации. Позволяет создавать компоненты, работающие с повышенными привилегиями, к которым обращаются обычные пользователи.

5. Управление Временем Жизни

DLL (In-Proc):

Загружается COM, когда клиент вызывает CoCreateInstance (или аналогичную функцию).

Выгружается только тогда, когда COM решит, что это безопасно. COM периодически вызывает экспортируемую из DLL функцию DllCanUnloadNow. DLL должна вернуть S\_OK, если все ее объекты уничтожены (глобальный счетчик ссылок DLL равен нулю), и тогда COM может выгрузить DLL из памяти клиента.

EXE (Out-Of-Proc):

Запускается COM, когда клиент вызывает CoCreateInstance.

Завершает работу сам, когда решит, что больше не нужен. Обычно это происходит, когда счетчик ссылок на все его активные объекты и фабрики классов падает до нуля. Сервер обнаруживает это и инициирует собственное завершение (например, завершает свой цикл сообщений или вызывает ExitProcess).

6. Передача Данных (Маршалинг)

DLL (In-Proc):

Если клиент и объект находятся в одном *апартаменте* (потоковой модели), маршалинг не нужен. Передаются прямые указатели на память.

Если клиент и объект в разных апартаментах (например, STA-клиент создает MTA-объект), COM автоматически использует прокси/стаб для маршалинга *внутри процесса* для безопасной передачи вызовов между потоками

EXE (Out-Of-Proc

Маршалинг всегда необходим, так как данные и вызовы пересекают границы процессов. COM использует стандартный маршалинг (прокси/стаб DLL, сгенерированную MIDL) или пользовательский маршалинг. Указатели на память не могут передаваться напрямую между процессами.

7. Потоковые Модели (Threading)

DLL (In-Proc):

Должен быть спроектирован с учетом потоковой модели клиента. В реестре для DLL указывается ThreadingModel (Apartment, Both, Free, Neutral). Неправильный выбор может привести к проблемам с производительностью или взаимным блокировкам.

EXE (Out-Of-Proc):

Работает в своем процессе со своими потоками и апартаментами. Взаимодействие с клиентом всегда идет через маршалинг, что несколько изолирует потоковые модели, но сам EXE-сервер все равно должен правильно управлять своими потоками и апартаментами.

8. Регистрация

DLL (In-Proc):

Регистрируется с помощью утилиты regsvr32.exe.

В реестре (в HKCR\CLSID\{...}) создается ключ InprocServer32, указывающий путь к DLL.

EXE (Out-Of-Proc):

Обычно регистрирует себя сам при запуске с параметром командной строки (например, /RegServer или -RegServer).

В реестре создается ключ LocalServer32, указывающий путь к EXE-файлу. Также часто регистрируется AppID для настройки безопасности и других параметров процесса.

Когда что использовать?

DLL (In-Proc):

Когда нужна максимальная производительность.

Когда не требуется изоляция от клиента или особые права доступа.

Для тесной интеграции (например, элементы управления ActiveX, обработчики оболочки Windows, плагины).

EXE (Out-Of-Proc):

Когда важна стабильность и изоляция от клиента.

Когда серверу нужн другие права доступа, чем у клиента.

Когда сервер должен обслуживать несколько клиентов и иметь независимое время жизни.

Когда сервер выполняет длительные фоновые задачи.

Когда допустимы накладные расходы на межпроцессное взаимодействие (например, серверы автоматизации вроде Excel, Word).

**DLL Сервер (In-Process)**

1. **Запуск:** Загружается операционной системой в адресное пространство клиентского процесса, когда клиент запрашивает объект из этой DLL.
2. **Предоставление Фабрик:**

* Не регистрирует фабрики при старте.
* COM вызывает экспортируемую из DLL функцию (DllGetClassObject), когда ему нужна фабрика классов для создания объекта.
* DLL создает экземпляр фабрики по требованию в ответ на вызов COM.

1. **Управление Жизнью Объектов:**

* Каждый объект имеет свой счетчик ссылок.
* Создание/уничтожение объекта (или вызов LockServer фабрикой) изменяет глобальный счетчик ссылок всей DLL.

1. **Управление Жизнью Сервера (Выгрузка):**

* Не завершает работу сам.
* COM периодически вызывает экспортируемую из DLL функцию (DllCanUnloadNow), чтобы спросить, можно ли ее выгрузить.
* DLL проверяет свой глобальный счетчик ссылок: если он 0, отвечает "Да" (S\_OK), и COM может выгрузить DLL.

1. **Регистрация:**

* Требует внешней утилиты (regsvr32.exe).
* regsvr32 вызывает экспортируемые из DLL функции (DllRegisterServer/DllUnregisterServer) для записи/удаления информации в реестре (путь к DLL и т.д.).

**EXE Сервер (Out-Of-Process)**

1. **Запуск**: Запускается COM как отдельный процесс, когда клиент запрашивает объект, а сервер еще не запущен. Выполняется его основная функция (main/WinMain).
2. **Предоставление Фабрик:**

* Сам создает экземпляры своих фабрик классов во время своего старта.
* Сам регистрирует эти фабрики в COM (CoRegisterClassObject), чтобы COM знал о них заранее.

1. **Управление Жизнью Объектов:**

* Каждый объект имеет свой счетчик ссылок.
* Создание/уничтожение объекта (или вызов LockServer фабрикой) изменяет глобальный счетчик блокировок всего EXE-сервера.

1. **Управление Жизнью Сервера (Завершение):**

* Сам отслеживает свой глобальный счетчик блокировок.
* Когда счетчик блокировок падает до нуля, сервер сам инициирует свое завершение (выходит из цикла сообщений, завершает процесс). COM не спрашивает его разрешения.

1. **Регистрация:**

* Обычно регистрирует сам себя.
* Обрабатывает аргументы командной строки (например, /RegServer) в своей main/WinMain для записи/удаления информации в реестре (путь к EXE и т.д.).