Магистерская программа 519200 - Открытые информационные системы, (специальная часть)

Вопрос 4.
Создание и применение COM-компонентов. COM-интерфейсы. COM-библиотека. Серверы автоматизации. Вызов функций из серверов автоматизации. Использование серверов автоматизации Word и Excel.

COM - это метод разработки программных компонентов - небольших двоичных исполняемых файлов, которые предоставляют необходимые **сервисы** приложениям, операционным системам и другим интерфейсам.
COM - это спецификация. Она указывает как создавать динамически взаимозаменяемые компоненты.
Разбивка монолитного приложения на компоненты делает приложение

- более динамичным,
- облегчает обновление частей приложения,
- позволяет собирать новые приложения из имеющихся частей - библиотеки компонентов,
- легче выполнить адаптацию приложения к...
- позволяет заменять компоненты во время работы приложения
- упрощение процесса разработки распределенных приложений.

Требования к COM компонентам

Компоненты должны:

- подключаться динамически
- инкапсулировать (скрывть) детали своей реализации.

Клиент - это программа или компонент, использующие другой компонент.
Клиент подсоединяется к компоненту через **интерфейс**.
Если компонент изменяется без изменения интерфейса, то изменения в клиенте не требуются.
Изоляция клиента (предоставляемого ему интерфейса) от реализации накладывает на компоненты **след.** ***ограничения***:

1. Компонент должен скрывать используемый язык программирования.
2. Компонент должен распространятся в двооичной форме.
3. Новые версии компонента должны работать как с новыми, та и со старыми клиентами.
4. Компоненты должны быть (прозрачно) перемещаемы по сети. Удаленный компоонент для клиента рассматривается также, как и локальный (иначе это бы вызывало перекомпиляцию клиента при перемещении компонента).
5. Компонент должен одинаково выполняться :
   - внутри одного процесса
   - в разных роцессах
   - на разных машинах.

Интерфейс COM - это не только набор функций, но и :
- это определенная структура в памяти,
- содержащая массив указателей на функции
- каждый элемент массива содержит адрес функции, реализуемой компонентом.

В C++ *чисто абстрактные* базовые классы - это базовые классы, содержащие только чисто виртуальные функции (virtual fx() = 0 ;).

Будем называть наследование от чисто абстрактного базового класса **- наследованием интерфейса.**
При использовании в C++ чисто абстрактных базовых классоов в памяти создается определенная структура - **таблица виртуальных функций**.
При программировании на других языках, например на Object Pascal, при объявлении интерфейса в памяти также памяти создается структура - таблица виртуальных функций.

В сзыке С ключевого слова interface не было предусмотрено, но Microsoft Win32 SDK соодержит заголовочный файл **OBJBASE.H**, в котором определен интерфейс:

```
#define interface struct
```

Язык программирования Object Pascal имеет ключевое слово interface, и любой определяемый интерфейс – это уже сразу COM интерфейс.
Соглашение о вызове функций: функции COM интерфейса должны быть определены как **_stdcall** - функция выбирает параметры из стека **перед возвратом в вызывающую процедуру**.

Любой COM интерфейс должен наследоваться от интерфейса IUnknown. IUnknown пределен в заголовочном файле UNKNWN.H Win32 SDK**.**

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

QueryInterface - метод, возвращающий указатель на запрашиваемый интерфейс. Требование **обязательного запроса интерфейса** делает систему прозрачной для изменения версий.

AddRef и **Release** – методы контролирующие количество ссылок на интерфейс. Если число ссылок на все интерфейчы компонента равно нулю, то компонент завершает свое выпололнение (управление количеством ссылок позволяет контролировать время жизни компонента).
Клиент знает компонента только через интерфейсы. Поэтому он не может напрямую управлять временем жизни копонента как такового, т.к.:

- в разных местах кода клиента могут быть вставлены вызовы этого компонента через различные интерфейсы и клиент может окончить использование одного интерфейса раньше другого;
- сложно определить момент удаления компонента из памяти, т.к. не ясно указывают ли два указателя на интерфейсы одного и того же компонента. Следовательно следует запрашиывать *IUnknown* через оба интерфейса и сравнивать результаты.
- ешение загрузить компоонент и не авгружать до конца выполнения программы не эффективно.
- выход - сообщать компоненту о начале использования каждого его интерфейса и о завершении, а компонент сам должен отслежживать число ссылок.
- сложность - понять отсутствует ли в программе ссылка ввиду оптимизации, или это ошибка.

Функции *AddRef* и *Release* описываются в компоненте и реализуют теккнику управления памятью, называемую *подсчет ссылок* (reference couting). Это позволяет компонентам самим себя удалять - когда значение счетчика доходит до нуля.
Существует три простых правила:

1. Функции, вызывающие нтерфейсы должны вызывать и AddRef для соответствующего указателя. (Это также относится и к QueryInterface и функции CreateInstance. Поэтому можно в клиенте не вызывать AddRef после получения указателя на интерфейс.)
2. При завершении работы с интерфейсом следует вызывать Release.
3. При создании новой ссылки на интерфейс (присвоении значения одного указателя другому) всегда следует вызывать AddRef.

Пример (IX – COM интерфейс, объявляющий функцию Fx, pIX – указатель на интерфейс IX)

Пример реализации метода QueryInterface:

```
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
        if (iid == IID_IUnknown)
        {               trace("QueryInterface: Return pointer to IUnknown.") ;
                *ppv = static_cast<IX*>(this) ;
        }
        else if (iid == IID_IX)
        {               trace("QueryInterface: Return pointer to IX.") ;
                *ppv = static_cast<IX*>(this) ;
        }
        else if (iid == IID_IY)
        {               trace("QueryInterface: Return pointer to IY.") ;
                *ppv = static_cast<IY*>(this) ;
        }
        else
        {
                trace("QueryInterface: Interface not supported.") ;
                *ppv = NULL ;
                return E_NOINTERFACE ;
        }
        reinterpret_cast<IUnknown*>(*ppv)->AddRef() ; // See Chapter 4.
        return S_OK ;
}
```

Соглашения по QueryInterface

- Вы всегда получаете оди и тот же IUnknown
- Вы сможете получить интерфейс снова, если смогли получить его раньше
- Вы можете снова получить интерфейс, который у вас еже есть
- Вы всегда можете вернуться туда, откуда начали (т.е., если IX получается через IY, то это возможно повторять)
- Если вы смогли попасть туда как-либо, то сможете попасть опять и по другому.

Интерфейсы, поддерживаемые компонентом - это те интерфейсы указатели на которые возвращает *QueryInterface*.

В модели DCOM используется QueryMultiInterface - для запроса нескольких интерфейсов за один вызов.

Вместо изменеия интерфейса - создание нового с новым IID.

--
Методы COM интерфейса возвражают значение типа **HRESULT.**
Все доступные коды возврата содержатся в файле Win32 WINERROR.H.
(#define E_NOINTERFACE 0x80004002L)
Код возврата - это 32 битовое значение:
| 31р - Признак критичности | 30-16рр - Средство| 15-0 Код возврата |
Код возвата:
S_OK
NOERROR
S_FALSE
E_UNEXPECTED неожиданная ошибка
E_NOTIMPL не реализовано
E_NOINTERFACE
E_OUTOFMEMORY
E_FAIL ошибка по неуказанной причине
…

Множественность кодов завершения
Желательно использовать макросы SUCCEEDED и FAILED, так как существует более одного кода успешного выполнений метода, и более одного кода завершения с ошибкой.

Каждый компонени и интерфейс регистрируется в реесте Windows со своим GUID.
GUID – это 128 битовое уникальное значение (48 - уникально ля компьютера и 60 - для времени) (**16** байтов).

COM библиотека

Для инициализации библиотеки COM процесс должен вызвать функцию:
HRESULT *CoInitialize* (void* reserved) // параметр равен NULL

.

При завершении работы - функцию: void *CoUninitialize()*.
Для каждого процесса библиотеку следует инициализировать только один раз. По общему соглашению COM инициализируется в EXE, а не в DLL.
Если библиотека уже была вызвана данным процессом, то она возвращает не S_OK, а S_FALSE.

Для создания COM компонента COM библиотека предоставляет функцию CoCreateInstance.

Эта функция библиотеки COM для создания других компонентов.

CoCreateInstance - функция COM- библиотеки (файл OLE32.DLL - для динамической компоновки или OLE32.LIB - для статической компоновки).

```
CoInitialize(NULL);  //Инициализация библиотеки
 HRESULT __stdcall CoCreateInstance(
        const CLSID& clsid,
        IUnkown* pIUnknownOuter,   // Внешний компонент
                                   // (используется только при агрегировании)
        DWORD dwCont, // Внешний контекст
        const IID& iid,  // Уникальный идентификатор запрошиваемого интерфейс
        void** ppv);     // Указатель на запрошиваемый интерфейс
CoUninitialize();    // Отключить библиотеку
```

Преимущества и недостатки различных контекстов: скорость и совместная память.
Контекст класса определяется следующими константами (или их совокупностью):

CLSCTX_INPROC_SERVER - Клиент принимает только компоненты, которые исполняются в одном с ним процессе
CLSCTX_LOCAL_SERVER Используемые компоненты выполняются в другом процессе (но на той же машине) - это EXE файлы.
CLSCTX_REMOTE_SERVER - допускаются компоненты, выполняющиес яна другой машиме (требуется DCOM)/

Файл OBJBASE.H содержит константы, позволяющие объединять контексты:
CLSCTX_INPROC
CLSCTX_ALL
CLSCTX_ SERVER

При программировании на Object Pascal для создания COM-компонентов можно использовать класс **TComObject** Основное отличие класса **TComObject** от класса **TInterfacedObject** в том, что он поддерживает фабрики классов.
Класс **TInterfacedObject** - это класс языка Object Pascal наследуемый от интерфейса IUnknown и реализующий его.
Для того чтобы создать любой COM-компонент следует или самостоятельно реализовать методы интерфейса IUnknown, или наследовать класс, который реализует эти методы.
Класс TInterfacedObject объявляется следующим образом:

```
type
 TInterfacedObject = class(TObject, IUnknown)
 private
  FRefCount: Integer;
 protected
  function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
 public
  property RefCount: Integer read FRefCount;
 end;
```

Перед использованием этих классов следует подключить COM библиотеку, указав:
uses ComObj;
Эта библиотека содержит метод CreateOleObject, который можно использовать для получения ссылки на компонент сервер автоматизации.

Диспетчерские интерфейсы и автоматизация

Автоматизация - другой способ управления компонентом. (Исп. для Word, Excel, Visual Basic, Java).
Автоматизация - надстройка над COM.

Сервер Автоматизации - это компонент COM, который реализует интерфейс **IDispatch**.
Контроллер Автоматизации - это клиент COM, взаимодействующий с сервером через интерфейс **IDispatch**. (Для вызовы функций сервера использует функции члены интерфейса - неявный вызов).
Первоначально Автоматизация разрабатывалась для Visual Basic.

Почти любой сервис, представимый через интерфейсы COM может быть представлени и через IDispatch.

IDispatch обепечивает для интерпретатора языка (исп-ие макроса для вызова функции компонента COM) вызов функции по трем параметрам:
- ProgID компонента
- имени функции и ее аргументов.

Описание интерфейса IDispatch

Этот интерфейс принимает имя функции и выполняет ее.

```
interface IDispatch : IUnknown                    //из файла OAidl.idl
  {
   HRESULT GetTypeInfoCount([out] UINT* pctinfo) ;

   HRESULT GetTypeInfo ([in] UINT iTInfo,
                [in] LCID lsid;;
                [out] ITypeInfo** ppTInfo) ;

HRESULT GetIDsOfName ([in] REFIID riid,          // Принимает имя функции и возвращает ее
                [in, size_is(cNames)] LPOLESTR *rgszNames,  // диспетчерский идентификатор
                [in] UINT cNames,
                [in] LCID lcid,
                [out, size_is(cNames)] DISPID *rgDispId) ;      // DISPID это длинное целое
                                                                // LONG и не уникально  [local]

//У каждой реализации IDispatch имеется свой собственный IID (иногда называется DIID).

HRESULT Invoke ([in] DISPID dispIdMember,    // Контроллер автоматизации передает
                [in] REFIID riid,                    // DISPID вызываемой функции в Invoke
                [in] LCID lsid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS* pDispParams,
                [out] VARIANT* pVarResult,
                [out] EXEPINFO* pExcepinfo,
                [out] UINT* puArgErr
                    );
```

DISPID используется функцией-членом Invoke как индекс в массиве указателей на функции.

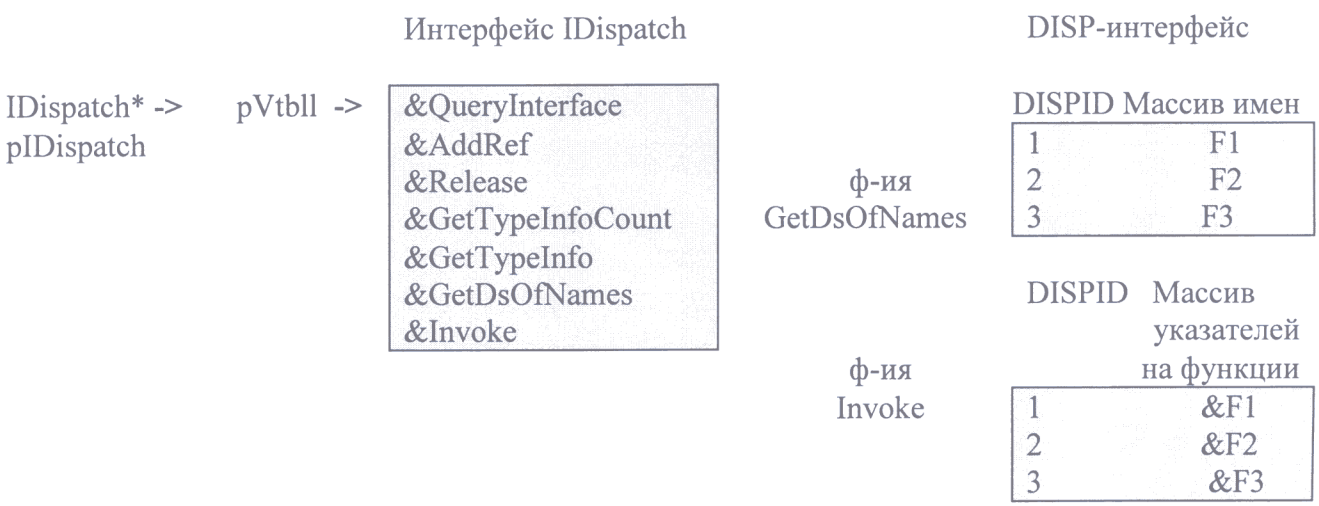
Однако сервер Автоматизации не обзан реализовывать Invoke именно таким образом. Он может использовать обычный ооператор switch.

DISP-интерфейсы

Набор функций, реализованных с помощью IDispatch::Invoke **называется диспетчерским интерфейсом (disp-интерфейсом)**

Реализация IDispatch::Invoke определяет набор функций, посредством которых взаимодействую сервер и контроллер автоматизации.

Пример возможной реализации DISP-интерфейса



Дуальные интерфейсы

Другой способ: интерфейс COM, реализующий IDispatch::Invoke наследует не *IUnknown*, а *IDispatch*. Это наз. дуальным интерфейсом.
Дуальный интерфейс - это disp-интерфейс все члены которого доступные через Invoke, доступны и напрямую через vtbl.

Для использования функции сервера автоматизации в контроллере автоматизации надо:

1. - создать компонент по его ProgID
2. - получить интерфейс IDispatch
3. - запросить DISPID функции (метод GetDsOfNames по имени функциии возвращает ее DISPID)
4. - выполнить Invoke для вызова функции (Invoke вызывает функцию по ее DISPID).

Использование серверов автоматизации Word и Excel.

Для использования сервера автоматизации следует подключить библиотеку типа, описывающую методы диспетчерского интерфейса.
Любая современная система программирования позволяет выполнить эту операццию (для Object Pascal в проект добавляется файл на языке Object Pascal собъявлением методов disp интерфейса, для C++ - в проект добавляется заголовочный файл с объявлением disp интерфейса, для C# - в проект добавляются соответствующие сборки.

Таким образом для использования серверов автоматизации Word и Excel в проект следует добавить информацию о disp интерфейса, реализуемым данным сервером автоматизации.

Например, при использовании языка программирования C#, следует добавить пространства имен:

```
using Microsoft.Office.Interop.Excel;
using Microsoft.Office.Interop.Word;
```

Для создания сервера автоматизации Word следует создать объект заданного типа. Например для создания и отображения сервера автоматизации Word, а затем добавления в него нового документа можно записать:

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
```

Для создания сервера автоматизации Excel и записи значения 123 в ячейку A1 можно записать:

```
var excelApp = new Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

Excel.Range targetRange = excelApp.Range["A1"];
targetRange.Value = "123";

excelApp.Range["B1"].Value = "444";
excelApp.Range["B1"].Select();
```

Для успешной работы с серверами автоматизации Word и Excel надо использовать **объектную модель документа**, реализуемую этими приложениями.
Так сервер автоматизации Excel содержит набор объектов, включая

- Application - представляет приложение Excel
- Workbook - представляет одну книгу в приложении Excel
- Worksheet - является членом коллекции листов Worksheets
- Range - представляет ячейку, строку, столбец и набор ячеек, содержащий один или несколько блоков ячеек (которые могут быть смежными или несмежными) или группу ячеек на нескольких листах

Excel предоставляет коллекцию Sheets как свойство объекта Workbook . Каждый элемент коллекции Sheets является объектом Worksheet или Chart .

Объектная модель точно соответствует пользовательскому интерфейсу.Объект Application представляет приложение в целом, и каждый из объектов Workbook содержит коллекцию объектов Worksheet. Отсюда следует, что основная абстракция, представляющая ячейки, является объектом Range, позволяющим работать с отдельными ячейками или группой ячеек.

Примеры использования:

```
//Для создания новой книги используйте метод Add из коллекции Workbooks.
var excelApp = new Excel.Application();
```

```
    Excel.Workbook newWorkbook =
    excelApp.Application.Workbooks.Add(missing);
// сохранение активной книги
excelApp.Application.ActiveWorkbook.Save();
```

В Framework .NET 4.0 (Visual Studio 10) значительно упрощено использование объектной модели, как как вместо методов set/get теперь можно использовать индексированные свойства (что позволяетиспользовать свойства типов COM с помощью стандартного синтаксиса C#)

Пример:

```
 // Visual C# 2010 excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
// Visual C# 2008
excelApp.get_Range("A1").Value2 = "ID";
excelApp.ActiveCell.get_Offset(1, 0).Select();
```

Еще одной особенностью мспользования серверов автоматизациии в В Framework .NET 4.0 является применение типа dynamic.

Тип **dynamic** позволяет пропускать проверки типов во время компиляции операции, в которых он применяется . Вместо этого эти операции разрешаются во время выполнения.Тип упрощает доступ к API модели COM, например API автоматизации Office.

Например, excelApp.Columns[1] возвращает Object, а метод AutoFit является методом класса Range в Excel. Без типа dynamic необходимо выполнять приведение объекта, возвращаемого excelApp.Columns[1], к экземпляру Range перед вызовом метода AutoFit.

```
    // В Visual C# 2008 требуется приведение типа
    ((Excel.Range)excelApp.Columns[1]).AutoFit();

    // В Visual C# 2010 приведение типа не нужно
    excelApp.Columns[1].AutoFit();
```