Capítulo

9

Interfaceando Matlab e Delphi através de Automation

Autor: Constantino Seixas Filho
UFMG – Departamento de Engenharia Eletrônica

Utilizando o Matlab a partir de um programa em Delphi

O primeiro passo é instanciar um objeto da classe da aplicação desejada, no nosso caso o Matlab. Inicialmente usaremos um método de fácil utilização, mas que apresenta alguns inconvenientes. Nós iremos definir uma variável do tipo **variant** e atribuir a ela um objeto COM suportando a interface **IDispatch**. Depois vamos usar este objeto para chamar métodos do servidor. O aspecto negativo deste tipo de enfoque é que ele é mais lento e proporciona uma checagem de tipos muito pobre. Toda a checagem de tipos envolvendo variants é feita em tempo de execução.

CreateOleObject

CreateOleObject irá criar um objeto da classe indicada e retornar uma **interface IDispatch**. Para usar a função inclua **ComObj** na lista **uses**.

```
function CreateOleObject(const ClassName: string): IDispatch;
```

O parâmetro ClassName especifica a **ClassID**.

Exemplo

```
Matlab := CreateOleObject('Matlab.Application');  // Cria instância de objeto
```

Matlab deve ser uma variável da classe **Variant**. O tipo **Variant** é compatível com quaisquer tipos básicos suportados por *automation*: inteiros, reais, booleanos, strings e arrays.

Um outra alternativa é usar a função *CreateComObject*, quando o Class ID é conhecido:

CreateComObject

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

Se o objeto não for liberado explicitamente ele permanecerá alocado.
Para fechar o Matlab basta fazer:

```
Matlab := Unassigned;   // Libera objeto
```

Para usar as funções disponibilizadas pela **interface** use:

```
Matlab.Execute('contour(a)');
```

Observe que *CreateOleObject* retorna um objeto do tipo IDispatch enquanto *CreateComObject* retorna um objeto do tipo IUnknown.

A função Execute irá executar qualquer comando do Matlab.

Para saber quais as funções disponibilizadas leia a seção seguinte: *Importando o Arquivo Type library do Matlab.*

Caso a aplicação já esteja em execução, a função a ser executada é:

```
function GetActiveOleObject(const ClassName: string): IDispatch;
```

Esta função retorna uma referência para uma interface **IDispatch** para um objeto COM registrado e em execução no momento da chamada.

Exemplo

```
MyWordApp := GetActiveOleObject( 'Word.Application' );
```

Exemplo Completo de Aplicação:

Este exemplo foi extraído de um forum de discussões do Delphi:

```
unit TestMatlab1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComObj;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    Matlab : Variant;
    zr, zi : OleVariant;
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  i, j : integer;
begin
    Caption := 'Matlab Magic';
    Matlab := CreateOleObject('Matlab.Application');  // Cria instância de objeto
    zr := VarArrayCreate([1, 16, 1, 16], varDouble);  // Cria array de Variants com double precision
    zi := VarArrayCreate([0, 0], varDouble);
    for i := 1 to 16 do
      for j := 1 to 16 do
        zr[i, j] := Random;   // Define a matriz zr (16, 16)
    Matlab.PutFullMatrix('a', 'base', VarArrayRef(zr), VarArrayRef(zi));
    Matlab.Execute('contour(a)');
    // Matlab := Unassigned;   // Libera objeto
end;

end.
```

Quando o botão da aplicação for acionado, observe o surgimento do ícone relativo ao Matlab na parte inferior da tela. Caso o ícone não apareça é porque a aplicação não foi encontrada no *registry*. Confira no registry o nome da aplicação, se ela suporta **Automation** e se o *path* para o executável está correto.

Importando o Arquivo *Type library* do Matlab

Existem duas maneiras de se conhecer os métodos de uma aplicação. Através da documentação do fornecedor e através da importação da *type library*.
O Delphi fornece ferramentas tanto para a visualização de uma *type library* como para transformar uma *type library* em uma unit Pascal. A vantagem de se importar uma *type library* é que cada método do servidor será exposto para o cliente e este poderá realizar uma checagem de tipos em tempo de compilação.

Visualização de uma *type library*

Para visualizar os métodos servidor do Matlab no Delphi faça:
Abra o Delphi e abra a pasta **mlapp.tlb** no diretório c:\Matlab\bin\Matlab.exe.
A seguinte janela será exibida:

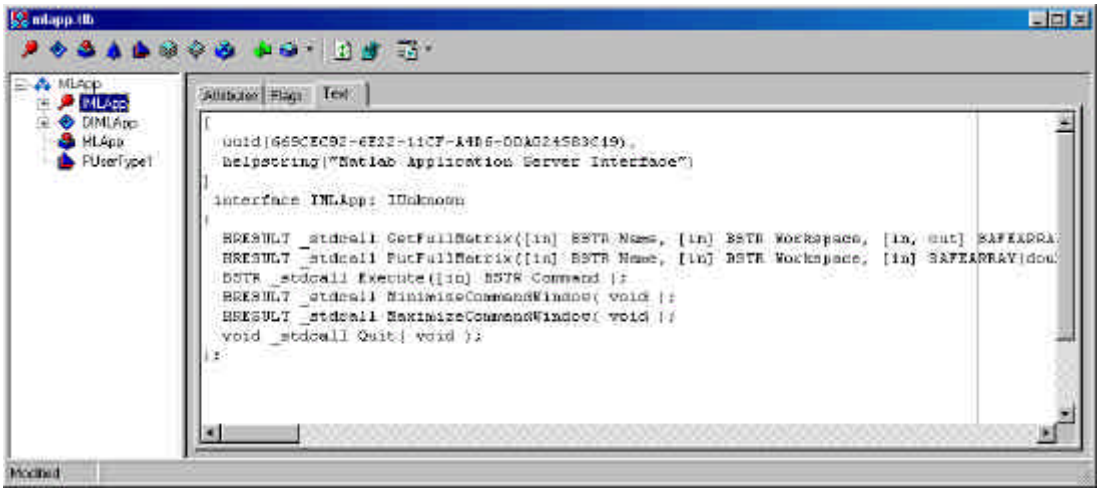


Figura 1 – Exibindo o conteúdo de uma *type library*

Selecione DIMLApp e o arquivo correspondendo à IDL em linguagem IDL será exibido:

```
[
  uuid(669CEC93-6E22-11CF-A4D6-00A024583C19),
  helpstring("Matlab Application Dispatch Interface")
]
 dispinterface DIMLApp
{
  properties:
  methods:
  [
  id(0x60000000),
  restricted
  ]
```

```
  void QueryInterface([in] GUID * riid, [out] void ** ppvObj );
  [
  id(0x60000001),
  restricted
  ]
  unsigned long AddRef( void );
  [
  id(0x60000002),
  restricted
  ]
  unsigned long Release( void );
  [
  id(0x60010000)
  ]
  void GetFullMatrix([in] BSTR Name, [in] BSTR Workspace, [in,
  out] SAFEARRAY(double) * pr,[in, out] SAFEARRAY(double) * pi );
  [
  id(0x60010001)
  ]
  void PutFullMatrix([in] BSTR Name, [in] BSTR Workspace, [in]
  SAFEARRAY(double) pr, [in] SAFEARRAY(double) pi );
  [
  id(0x60010002)
  ]
  BSTR Execute([in] BSTR Command );
  [
  id(0x60010003)
  ]
  void MinimizeCommandWindow( void );
  [
  id(0x60010004)
  ]
  void MaximizeCommandWindow( void );
  [
  id(0x60010005)
  ]
  void Quit( void );
};
```

No ambiente de desenvolvimento do Delphi é possível selecionar como visualizar *type libraries*, se na linguagem IDL que é o default, ou em Pascal. Isto é selecionado no menu Tools> Environment Options. Ao gerar uma *type library* de um servidor COM desenvolvido em Delphi, o ambiente Delphi permite escolher entre duas linguagens: IDL que é o default ou Pascal (ver Figura 2).

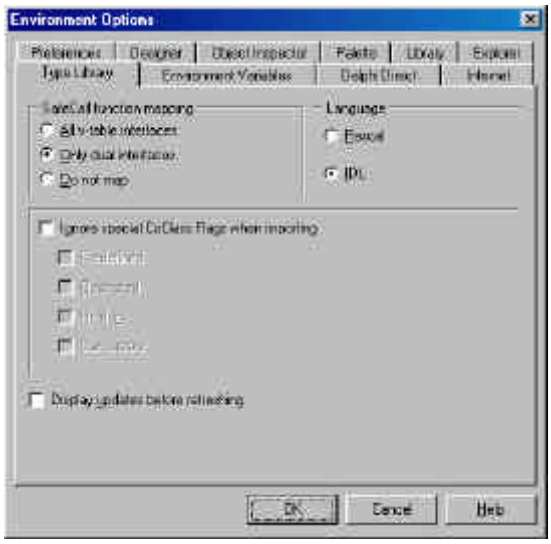


Figura 2 – Configuração do ambiente Delphi: *type library* em IDL ou Pascal

Ao se acionar o botão refresh na barra superior da janela a *type library* em formato Pascal será gerada no diretório c:\Program Files\Borland \Delphi6\Imports.

Um trecho deste arquivo é mostrado a seguir:

```
unit MLApp_TLB;

// ************************************************************************ //
// WARNING
// The types declared in this file were generated from data read from a  Type Library.
// If this type library is explicitly or indirectly (via another type library referring to this type
// library) re-imported, or the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost

// ************************************************************************ //
// PASTLWTR : $Revision:   1.130  $
// File generated on 28/05/03 22:41:48 from Type Library described below.
// ************************************************************************  //

// Type Lib: C:\MATLAB\bin\mlapp.tlb (1)
// LIBID: {669CEC91-6E22-11CF-A4D6-00A024583C19}
// LCID: 0
// Helpfile:
// DepndLst:
//   (1) v1.0 stdole, (C:\WINDOWS\SYSTEM\stdole32.tlb)
//   (2) v2.0 StdType, (c:\WINDOWS\SYSTEM\OLEPRO32.DLL)
//   (3) v1.0 StdVCL, (c:\WINDOWS\SYSTEM\stdvcl32.dll)
// Errors:
//   Hint: TypeInfo 'MLApp' changed to 'MLApp_'

// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}

interface

uses ActiveX, Classes, Graphics, OleServer, StdVCL, Variants, Windows;
```

```
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx

const
  // TypeLibrary Major and minor versions
  MLAppMajorVersion = 1;
  MLAppMinorVersion = 0;

  LIBID_MLApp: TGUID = '{669CEC91-6E22-11CF-A4D6-00A024583C19}';

  mpo: TGUID = '{669CEC92-6E22-11CF-A4D6-00A024583C19}';
  DIID_DIMLApp: TGUID = '{669CEC93-6E22-11CF-A4D6-00A024583C19}';
  CLASS_MLApp_: TGUID = '{669CEC94-6E22-11CF-A4D6-00A024583C19}';

type
// Forward declaration of types defined in TypeLibrary
  IMLApp = interface;
  DIMLApp = dispinterface;

// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
  MLApp_ = DIMLApp;

// Declaration of structures, unions and aliases.
  PUserType1 = ^TGUID; {*}

// Interface: IMLApp
// Flags:     (0)
// GUID:      {669CEC92-6E22-11CF-A4D6-00A024583C19}
  IMLApp = interface(IUnknown)
    ['{669CEC92-6E22-11CF-A4D6-00A024583C19}']
    function  GetFullMatrix(const Name: WideString; const Workspace: WideString;
                        var pr: PSafeArray; var pi: PSafeArray): HResult; stdcall;
    function  PutFullMatrix(const Name: WideString; const Workspace: WideString; pr:
PSafeArray;
                        pi: PSafeArray): HResult; stdcall;
    function  Execute(const Command: WideString): WideString; stdcall;
    function  MinimizeCommandWindow: HResult; stdcall;
    function  MaximizeCommandWindow: HResult; stdcall;
    procedure Quit; stdcall;
  end;

// DispIntf:  DIMLApp
// Flags:     (4096) Dispatchable
// GUID:      {669CEC93-6E22-11CF-A4D6-00A024583C19}
  DIMLApp = dispinterface
    ['{669CEC93-6E22-11CF-A4D6-00A024583C19}']
    procedure MaximizeCommandWindow; dispid 1610678276;
    procedure Quit; dispid 1610678277;
  end;
// Nota Prof: Observe que os dispids estão em decimal e não em hexa como no arquivo original

// The Class CoMLApp_ provides a Create and CreateRemote method to create instances of the
// default interface DIMLApp exposed by the CoClass MLApp_. The functions are intended to be
//used by clients wishing to automate the CoClass objects exposed by the server of  this typelibrary
  CoMLApp_ = class
```

```
    class function Create: DIMLApp;
    class function CreateRemote(const MachineName: string): DIMLApp;
  end;

implementation

uses ComObj;

// Nota Prof: Cria um objeto do tipo do servidor localmente

class function CoMLApp_.Create: DIMLApp;
begin
  Result := CreateComObject(CLASS_MLApp_) as DIMLApp;
end;

// Nota Prof: Cria um objeto do tipo do servidor remotamente

class function CoMLApp_.CreateRemote(const MachineName: string): DIMLApp;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MLApp_) as DIMLApp;
end;

end.
```

Observe que o Matlab 5 só suporta a criação de servidores do tipo **dispinterface**. Não é possível se criar um componente servidor IMLApp.

Quanto a velocidade de uso haveria a seguinte ordem de preferência caso o Matlab suportasse a interface *raw* (IMLApp):

a) Uso da interface IMLApp (MatalabServer: IMLApp).
b) Uso da interface DIMLApp (MatlabServer: DIMLApp)
c) Uso de variants (MatlabServer: Variant)

Como escrever um Ole client control importando uma *type library*

Type libraries são arquivos binários que tanto podem ser arquivos isolados com a extensão .tlb ou .olb como podem ser arquivos .exe ou .dll.

Para importar uma *type library* gerada em outro programa, no Delphi, chame **Project**>**ImportTypeLibrary**. Se a *type library* já estiver no menu, clique o nome da *type library* e **CreateUnit**. É melhor desativar o flag **Generate Component Wrapper**, ou o componente gerado deverá ser carregado pelo usuário que receber o aplicativo que use o componente.

Se a lista não contiver a *type library* desejada clique **Add** e escolha o diretório onde existe o arquivo .tlb.

No caso do Matlab o diretório c:\Matlab\bin\Matlab.exe contém a *type library* mlapp.tlb. Após acionar **Add** a janela da Figura 3 será exibida.

Delphi irá criar o arquivo MLApp_TLB.pas no diretório c:\Program Files\Borland \Delphi6\Imports que define a unit MLApp_TLB.

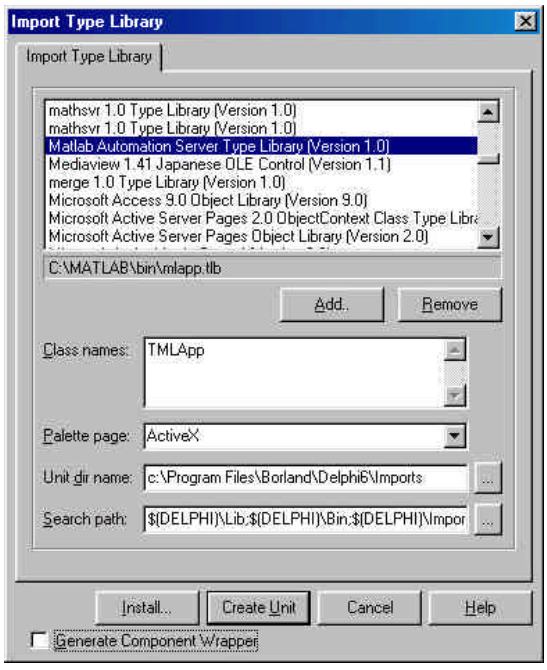


Figura 3 – Janela mostrando a *type libray* do Matlab instalada

Clique **CreateUnit** e o programa **MLApp_TLB.pas** será criado.

Na Unit1 inclua a Unit MLApp_TLB na lista de **uses**:

uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

```
Dialogs, MLApp_TLB;
```

- No **Form1** crie um botão de *label* Teste.
- Associe um evento para à ação de *OnClick* do botão clicando duas vezes sobre o botão.

O cliente tem duas maneiras de chamar um método do servidor: utilizando uma **interface** *custom* ou *raw*, isto é uma VTable ou utilizando uma *dispinterface* (*dispatch interface*) que basicamente mapeia cada entrada de uma interface em um número (*dispid*). Este artificio foi criado para permitir o acesso a servidores por programas escritos em uma linguagem que não suporta ponteiros para funções e portanto não pode usar uma VTable.

- No corpo do evento declare uma variável de nome MatlabServer do tipo **DIMLApp ou Variant**,

```
var
  Form1: TForm1;

  // Uso da Interface Direta: Mais Rápido, mas não suportdo por Matlab 5
  MatlabServer1: IMLApp;       // mostra parâmetros da função da classe na digitação
  // Uso do Variant: Mais lento
  MatlabServer2: Variant;      // não mostra parâmetros da função da classe na digitação
  // Quase tão rápido quanto método 1
  MatlabServer3: DIMLapp;      // mostra parâmetros da função da classe na digitação
```

Ao digitar MatlabServer1. uma janela com as opções de funções será mostrada. Escolha a função desejada.

Figura 4 – Help automático com rotinas disponíveis na Interface IMLApp

Ao digitar MatlabServer3. uma janela com as opções de funções será mostrada. Escolha a função desejada.

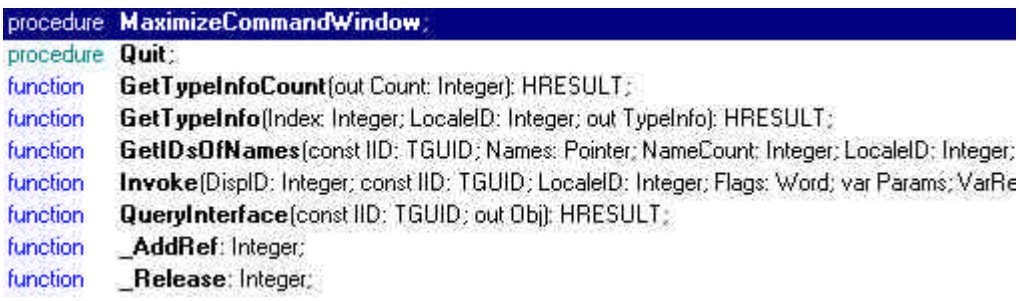


Figura 5 – Help para interface DIMLApp

Observe que apenas duas das funções da interface foram importadas.

No handler o evento click do botão coloque o código desejado:

```
procedure TForm1.DispInterfClick(Sender: TObject);
begin
  MatlabServer3 := CoMLApp_.Create;      // Cria um objeto servidor
  MatlabServer3.MaximizeCommandWindow;
end;
```

Ao clicar o botão o servidor é criado, mas a chamada da função *MaximizeCommandWindow* gera uma exception. Este problema foi repassado ao forum de discussão do Matlab.

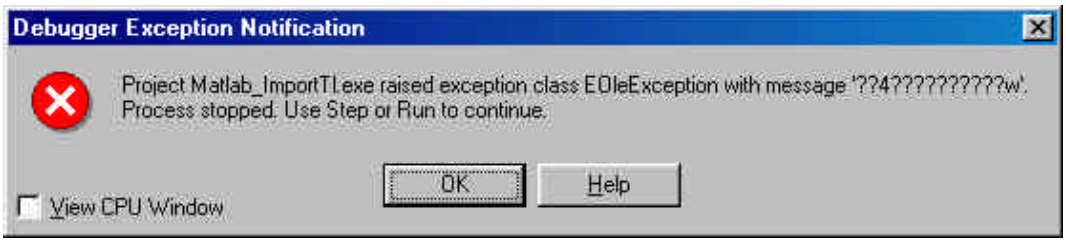


Figura 6 – Erro reportado pelo Delphi

Superando o problema

Embora não conhecendo a causa do problema, foi possível supera-lo da seguinte forma:
O arquivo MLApp_TLB.pas foi editado manualmente e as funções faltantes foram introduzidas. Por algum motivo o tipo PSafeArray não foi aceito e foi trocado para variant. Os dispids faltantes foram completados.

```
/ ************************************************************************//
// DispIntf:  DIMLApp
// Flags:     (4096) Dispatchable
// GUID:      {669CEC93-6E22-11CF-A4D6-00A024583C19}
// ************************************************************************//
  DIMLApp = dispinterface   // Modificada manualmente pelo Professor
    ['{669CEC93-6E22-11CF-A4D6-00A024583C19}']
    function GetFullMatrix(const Name: WideString; const Workspace: WideString;
                           var pr: Variant; var pi: Variant): HResult;  dispid 1610678272;
    function PutFullMatrix(const Name: WideString; const Workspace: WideString;
                           pr: Variant; pi: Variant): HResult; dispid 1610678273;
    function Execute(const Command: WideString): WideString; dispid 1610678274;
    procedure MinimizeCommandWindow; dispid 1610678275;
    procedure MaximizeCommandWindow; dispid 1610678276;
    procedure Quit; dispid 1610678277;
  end;
```

O programa funcionou perfeitamente exceto a função *MinimizeCommandWindow* e *MaximizeCommandWindow*, sem causa aparente.

O código fonte do programa é dado a seguir:

```
var
  Form1: TForm1;

   // Uso da Interface Direta: Mais Rápido, mas não suportdopor Matlab 5
```

```
  MatlabServer1: IMLApp;       // mostra parâmetros da função da classe na digitação
  // Uso do Variant: Mais lento
  MatlabServer2: Variant;      // não mostra parâmetros da função da classe na digitação
  // Quase tão rápido quanto método 1
  MatlabServer3: DIMLapp;      // mostra parâmetros da função da classe na digitação

implementation

{$R *.dfm}

procedure TForm1.InterfDirClick(Sender: TObject);
begin
   MessageDlg ('Não suportado pelo Matlab5', mtError, [mbOK], 0);
end;


procedure TForm1.VariantIntClick(Sender: TObject);
begin
   MatlabServer2:= CoMLApp_.Create;       // Cria um objeto servidor
  try
    MatlabServer2.Execute('helpwin');
    MatlabServer2.MinimizeCommandWindow;   // Não funciona por motivo desconhecido
  except
    on EOleException do MessageDlg ('Erro na função de automação: ', mtError, [mbOK], 0);
    on E: Exception do
    begin
       MessageDlg ('Exceção Detectada: ' + E.Message, mtError, [mbOK], 0);
       //raise
    end;
  else MessageDlg ('Ops: ', mtError, [mbOK], 0);
  end; // end except
end;


procedure TForm1.DispInterfClick(Sender: TObject);
var
  i, j : integer;
  zr, zi : OleVariant;
begin
  MatlabServer3:= CoMLApp_.Create;      // Cria um objeto servidor
  zr := VarArrayCreate([1, 16, 1, 16], varDouble);  // Cria array de Variants  tipo double precision
  zi := VarArrayCreate([0, 0], varDouble);
  for i := 1 to 16 do
    for j := 1 to 16 do
      zr[i, j] := Random;   // Define a matriz zr (16, 16)

  MatlabServer3.PutFullMatrix('a', 'base', VarArrayRef(zr), VarArrayRef(zi));
  MatlabServer3.Execute('contour(a)');
end;

end.
```

Documentação oficial de Automação do Matlab

MATLAB COM Automation Methods

This section lists the methods that are supported by the MATLAB Automation Server. The data types for the arguments and return values are expressed as Automation data types, which are language-independent types defined by the Automation protocol. For example, BSTR is a wide-character string type defined as an Automation type, and is the same data format used by Visual Basic to store strings. Any COM-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

BSTR Execute([in] BSTR Command);

This command accepts a single string (`Command`), which contains any command that can be typed at the MATLAB command window prompt. MATLAB will execute the command and return the results as a string. Any figure windows generated by the command are displayed on the screen as if the command were executed directly from the command window or an M-file. A Visual Basic example is

```
Dim MatLab As Object
Dim Result As String
Set MatLab = CreateObject("Matlab.Application")
Result = MatLab.Execute("surf(peaks)")
```

void GetFullMatrix([in] BSTR Name, [in] BSTR Workspace, [in, out] SAFEARRAY(double)* pr, [in, out] SAFEARRAY(double)* pi);

> **Note** The first statement above should be declared in the general declarations section in order to keep the scope throughout the application.

This method retrieves a full, one- or two-dimensional real or imaginary `mxArray` from the named workspace. The real and (optional) imaginary parts are retrieved into separate arrays of doubles.

Name. Identifies the name of the `mxArray` to be retrieved.

Workspace. Identifies the workspace that contains the `mxArray`. Use the workspace name "base" to retrieve an `mxArray` from the default MATLAB workspace. Use the workspace name "global" to put the `mxArray` into the global MATLAB workspace. The "caller" workspace does not have any context in the API when used outside of MEX-files.

pr. Array of reals that is dimensioned to be the same size as the `mxArray` being retrieved. On return, this array will contain the real values of the `mxArray`.

pi. Array of reals that is dimensioned to be the same size as the `mxArray` being retrieved. On return, this array will contain the imaginary values of the `mxArray`. If the requested `mxArray` is not complex, an empty array must be passed. In Visual Basic, an empty array is declared as
`Dim Mempty() as Double`. A Visual Basic example of this method is

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag() As Double
Dim RealValue As Double
Dim i, j As Integer
rem We assume that the connection to MATLAB exists.
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8;]")
Call MatLab.GetFullMatrix("a", "base", MReal, MImag)
For i = 0 To 1
    For j = 0 To 3
    RealValue = MReal(i, j)
    Next j
Next i
```

void PutFullMatrix(
[in] BSTR Name,
[in] BSTR Workspace,
[in] SAFEARRAY(double) pr,
[in] SAFEARRAY(double) pi);

> **Note** The first statement above should be declared in the general declarations section in order to keep the scope throughout the application.

This method puts a full, one- or two-dimensional real or imaginary `mxArray` into the named workspace. The real and (optional) imaginary parts are passed in through separate arrays of doubles.

Name. Identifies the name of the `mxArray` to be placed.

Workspace. Identifies the workspace into which the `mxArray` should be placed. Use the workspace name "base" to put the `mxArray` into the default MATLAB workspace. Use the workspace name "global" to put the `mxArray` into the global MATLAB workspace. The "caller" workspace does not have any context in the API when used outside of MEX-files.

pr. Array of reals that contains the real values for the `mxArray`.

pi. Array of reals that contains the imaginary values for the `mxArray`. If the `mxArray` that is being sent is not complex, an empty array must be passed for this parameter. In Visual Basic, an empty array is declared as `Dim Mempty() as Double`. A Visual Basic example of this method is

```
Dim MatLab As Object
Dim MReal(1, 3) As Double
```

```
Dim MImag() As Double
Dim i, j As Integer
For i = 0 To 1
    For j = 0 To 3
    MReal(i, j)  = I * j;
    Next j
Next I
rem We assume that the connection to MATLAB exists.
Call MatLab.PutFullMatrix("a", "base", MReal, MImag)
```

> **Note** The first statement above should be declared in the general declarations section in order to keep the scope throughout the application.

MATLAB Automation Properties

You have the option of making your server application visible or not by setting the `Visible` property. When visible, the server window appears on the desktop, enabling the user to interact with the server application. This may be useful for such purposes as debugging. When not visible, the server window does not appear, thus perhaps making for a cleaner interface and also preventing any interaction with the server application.

By default, the `Visible` property is enabled, or set to `1`:

```
h = actxserver('Matlab.Application');
get(h, 'visible')
ans =
     1

```

You can change the setting of `Visible` by setting it to `0` (invisible) or `1` (visible). The following command removes the server application window from the desktop:

```
set(h, 'visible', 0);

get(h, 'visible')
ans =
     0

```

Additional Automation Server Information

Launching the MATLAB Server

For MATLAB to act as an Automation server, it must be started with the /`Automation` command line argument. Microsoft Windows does this automatically when a COM connection is established by a controller. However, if MATLAB is already running and was launched *without* this parameter, any request by an Automation controller to connect to MATLAB as a server will cause Windows to launch another instance of MATLAB with the `/Automation` parameter. This protects controllers from interfering with any interactive MATLAB sessions that may be running.

Specifying a Shared or Dedicated Server

You can launch the MATLAB Automation server in one of two modes -- shared or dedicated. A dedicated server is dedicated to a single client; a shared server is shared by multiple clients.

The mode is determined by the Program ID (ProgID) used by the client to launch MATLAB. The ProgID, `Matlab.Application`, specifies the default mode, which is shared. You can also use the version-specific ProgID, `Matlab.Application.N`, where `N` is equal to the major version of MATLAB you are running, (for example, `N` = 6 for MATLAB 6.5).

To specify a dedicated server, use the ProgID, `Matlab.Application.Single`, (or the version-specific ProgID, `Matlab.Application.Single.N`). The term `Single` refers to the number of clients that may connect to this server.

Once MATLAB is launched as a shared server, all clients that request a connection to MATLAB by using the shared server ProgID will connect to the already running instance of MATLAB. In other words, there is never more than one instance of a shared server running, since it is shared by all clients that use the ProgID that specifies a shared server.

> **Note** Clients will not connect to an interactive instance of MATLAB, that is, an instance of MATLAB that was launched manually and without the /`Automation` command line flag.

Each client that requests a connection to MATLAB using a dedicated ProgID will cause a separate instance of MATLAB to be launched, and that server will not be shared with any other client. Therefore, there can be several instances of a dedicated server running simultaneously, since the dedicated server is not shared by multiple clients.

Using MATLAB as a DCOM Server

DCOM is a protocol that allows COM connections to be established over a network. If you are using a version of Windows that supports DCOM (Windows NT 4.0 at the time of this writing) and a controller that supports DCOM, you can use the controller to launch MATLAB on a remote machine. To do this, DCOM must be configured properly, and MATLAB must be installed on each machine that is used as a client or server. (Even though the client machine will not be running MATLAB in such a configuration, the client machine must have a MATLAB installation because certain MATLAB components are required to establish the remote connection.) Consult the DCOM documentation for how to configure DCOM for your environment.

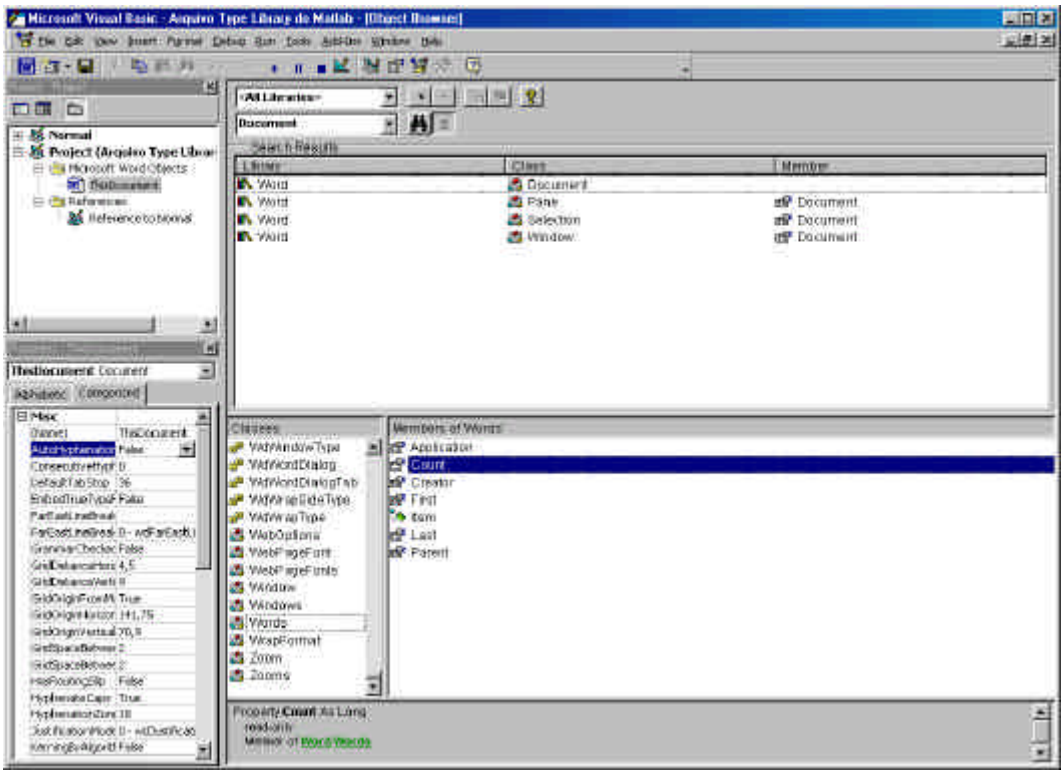
Exercícios

1) Abra o registry e localize a aplicação Matlab no HKEY_CLASSES_ROOT. Anote o seu CLSID. Agora pagine o item CLSID até encontrar o valor anotado. Verifique o *path* para a aplicação.

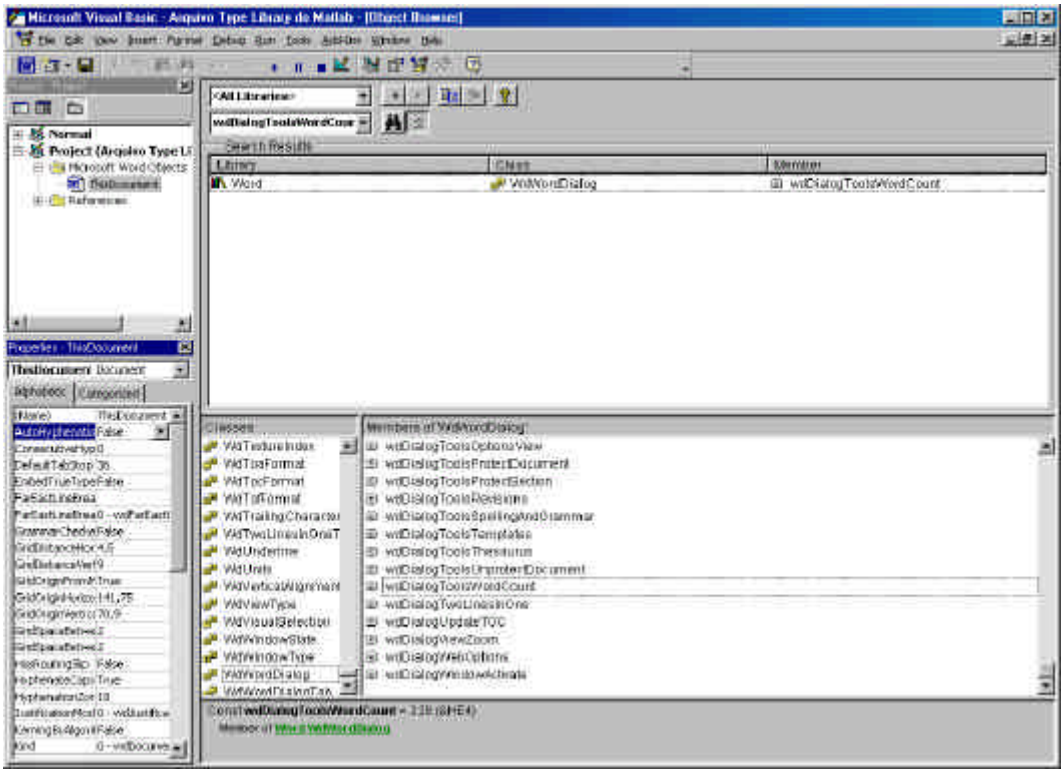
2) Crie uma aplicação em Delphi. Na janela crie um botão e clique duas vezes nele para criar um *handler* para o evento. Use o seguinte código para o *Handler*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyVar : Variant;
begin
      MyVar := CreateOleObject('Word.Application');
      MyVar.Visible := true;  // Mantem o Word aberto
end;
```

3) Ao automatizar aplicações baseadas no Word, invoque o *Visual Basic Object Browser*, acionando <ALT> <F11> de dentro do Word. Você vai enxergar todos os objetos disponíveis para o Word. Acione <F2> e objeto browser irá aparecer. Selecione a classe Document no painel superior, depois escolha a classe Words na coluna da esquerda do painel inferior e depois o membro Count. Qual o tipo desta propriedade ? Qual a sua permissão ? R/W ou *Read Only* ?



4) Ainda usando o browser procure o valor da constante:
wdDialogToolsWordCount.



5) Ache o arquivo Msword9.olb e visualize seu conteúdo no Delphi.

6) Importe a *type library* Msword9.olb para o Delphi. O arquivo Word_TLB.pas será criado no diretório C:\Program Files\Borland\Delphi6\Imports. Abra este arquivo e veja seu conteúdo. Procure a constante wdDialogToolsWordCount. Qual o seu valor ? Adicione este arquivo a lista de uses de seu aplicativo.

7) No programa abaixo, o compilador não pode verificar se o nome do método está correto em tempo de compilação. Troque o nome Insert por Isnsert e execute novamente o programa.

```
var
   VarW: Variant;
begin
     VarW := CreateOleObject (‘Word.Basic’);
     VarW.FileNew;
     VarW.Insert (‘Mastering Delphi by Marco Cantù’);
```

8) Desenvolva uma aplicação em Delphi que formate um relatório de alarmes em formato Word (.doc). Use cores para enfatizar o estado de alarme do estado de normalidade. Por exemplo: pontos em alarme devem ser descritos em vermelho. Pontos reconhecidos, mas em alarme em amarelo e pontos normalizados em verde.

9) Marque V ou F:

( ) Word, Excel e Matlab são exemplos de *Ole Automation Servers*.
( ) O cliente é também chamado de *Ole controller*.
( ) Uma *type library* é independente de linguagem.
( ) Variants têm seu tipo checado em tempo de execução (*run-time*).
( ) O *dispid* é um número que mapeia uma entrada de uma *dispinterface*.
( ) *dispinterface* utiliza *late binding* enquanto VTables possibilitam *early binding*.
( ) Chamar uma função pelo seu nome e fazer a consistência de tipo em tempo de execução é mais lento do que usar *dispinterfaces* que checam o tipo em tempo de compilação que por sua vez é mais lento que o uso de VTables.
( ) Se você usou a função *CreateOleObject* para partir uma aplicação e ela não foi encontrada isto pode indicar que o *path* para aplicação está errado. O caminho a seguir é conferir o nome da aplicação e o seu *path* no registry.
( ) A classe CoMLApp_ tem como função criar um objeto servidor, de acordo com o estabelecido na *type library*.
( ) Uma outra vantagem de se adotar *early binding* é que o nome dos métodos da interface são apresentados automaticamente durante a digitação do programa no editor do Delphi.
( ) Quando utilizamos *late binding*, se a aplicação especificada com *CreateOleObject* parte, mas a função pedida não é encontrada, isto aponta para um problema na *type library*, pois o nome da função é procurado em tempo de execução neste arquivo.

Bibliografia

[Cantù 2001] Marco Cantù, Mastering Delphi 6, Sybex Inc, 2001

[Collingbourne Mar2001] Huw Collingbourne, How to automate alternative applications using Delphi, PCPlus.co.uk, nº 174, March 2001, www.pcplus.co.uk/media/pcplus/pdf/174/174.progworld.delphi.pdf consultado em 31/05/2003

[Collingbourne Apr2001] Huw Collingbourne, Importing type libraries to control external applications, PCPlus.co.uk, nº 175, April 2001, www.pcplus.co.uk/media/pcplus/pdf/175/175.progworld.delphi.pdf consultado em 31/05/2003