



maxbox Starter 22

Start with Services

1.1 From DLL to COM

COM clients are applications that make use of a COM object or service implemented by another application or library. The most common types are applications that control an Automation server (Automation controllers) and applications that host an ActiveX control (ActiveX containers). Each service is an interface that presents a set of related functions. The implementation of the interface is hidden within the object.

As you will see a service and other interfaces fall into two basic categories. Hope you did already read Starters 1 till 21 at:

<http://sourceforge.net/apps/mediawiki/maxbox/>

In this lesson we deal with OleObjects and his creator.

`CreateOleObject` creates a single uninitialized object of the class specified by the `ClassName` parameter. `ClassName` specifies the string representation of the Class ID (CLSID).

`CreateOleObject` is used to create an object of a specified type when the CLSID is known and when the object is on a local or in-proc server. Only the objects that are not part of an aggregate are created using `CreateOleObject`.

In a late binding system like that, a type is unknown till runtime. So we use a general type or intrinsic type called variant. What's a Variant?

A Variant type is capable of representing values that change type dynamically. Whereas a variable of any other type is statically bound to that type, a variable of Variant type can assume values of differing types at run-time.

☞ The Variant type is most commonly used in situations where the actual type to be operated upon varies or is unknown at compile-time.

While variants offer great flexibility, they also consume more memory than regular variables, and operations on variants are substantially slower than operations on statically typed values.

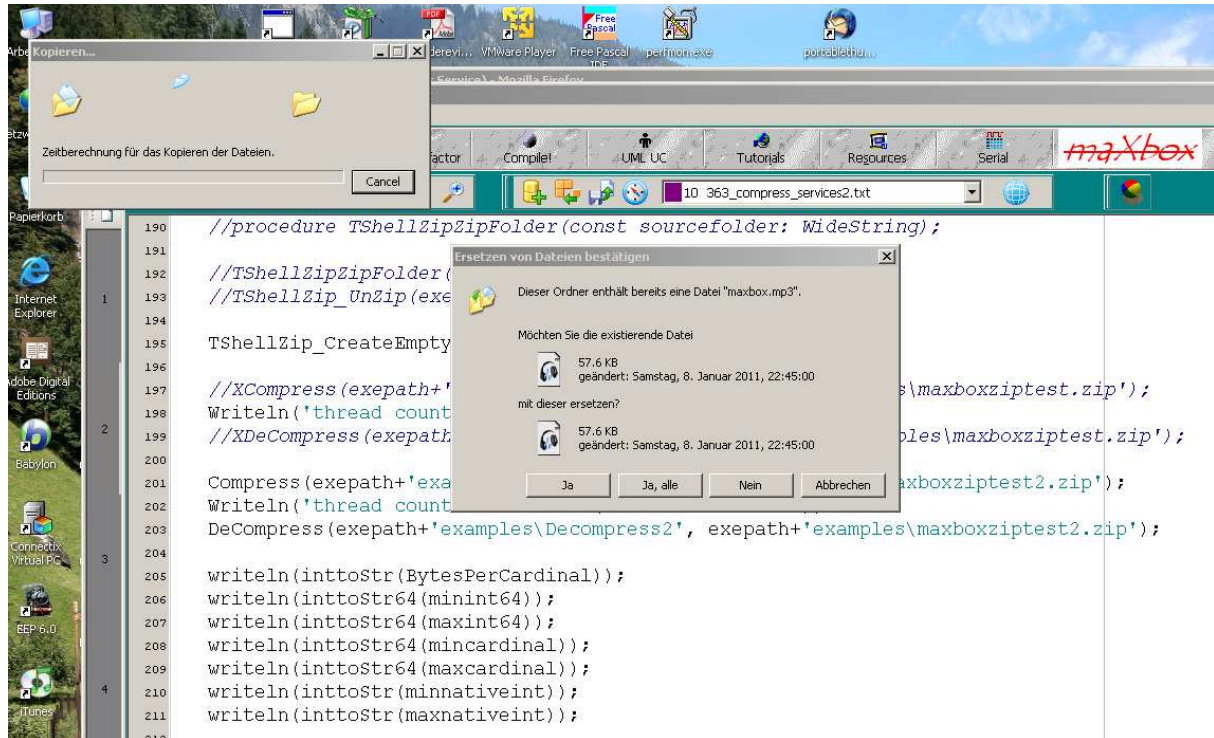
☞ An advantage of using Variants is that you do not need to import the type library, because Variants use only the standard `IDispatch` methods to call the server. The trade-off is that Variants are slower, because they use dynamic binding at runtime.

In this tutorial we show 3 steps to build a service:

- First building block is the use of the unit code in the script.
- Second we wrap the unit code to built a class unit of the shell code.
- Third we set two service procedures (compress and decompress) to call the class methods of the unit, it's very powerful, but tough to built.

When I start a service through another one small UI like a button the service has started successfully and it is running in the machine. It is running even I restart the machine also. But the code (Start method of Compress) is executed and takes some time, depending on the size of the archive, so a thread or more runs in the background.

This will perform compress operation (read and write data) and if records are found then it will Pause the current thread (i.e. Main/Parent thread) and start MultiThreading¹ to perform some copy operation, like the following screen-shot figure out.



1: Compress runs

Let's jump to history and the beginning of OLE with DDE. Dynamic Data Exchange was first introduced in 1987 with the release of Windows 2.0 as a method of interprocess communication so that one program can communicate with or control another program, somewhat like Sun's RPC (Remote Procedure Call).

In Delphi code, `CreateOleObject` is called once to create each new single instance of a class. To create multiple instances of the same class, we recommend that you use a class factory for example:

```
says TComponentFactory.Create(...) with
TPacketInterceptFactory.Create(...).
```

This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server create an object instance for the client.

¹ You can pause/stop the Main Thread and start again, once MultiThreading task gets completed.

1.2 Compress and Decompress

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom or the interface part on the right. Change that in the menu /View at our own style.



In maXbox you will execute the Compress as a script, libraries and units are already built.



Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from <http://www.softwareschule.ch/maxbox.htm> (you'll find the download maxbox3.zip on the top left of the page). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox3.exe` the box opens a default demo program. Test it with F9 / F2 or press **Compile** and you should hear a sound. So far so good now we'll open the example:

```
363_compress_services2.txt
```

File size: 15572

If you can't find the two files try also the zip-file (15572 bytes) loaded from:

http://www.softwareschule.ch/examples/363_compress_services2.txt

Now let's take a look at the code of this fist part project. Our first line is

```
01 program CompressServices2;
```

We name it, means the program's name is above.



This example requires two objects from the classes: `TShellZip` and `TMemoryStream` of `mX4` so the second one is from the well known `VCL Lib`.

Let's do a first Compress now. We want to check if our service is valid on your operating system. We use straight forward the function in the box:

```
201 Compress(exepath+'examples\earthplay2',
exepath+'examples\maxboxziptest2.zip');
```

Those are useful global procedures:

```
8488: Procedure Compress(azipfolder, azipfile: string);
8489: Procedure DeCompress(azipfolder, azipfile: string);
```

It is valid if a string `azipfolder` matches what the file system expects.



If its not find a valid folder or file to compress it says:

```
>>> Variant is null, cannot invoke.
```

So the magic behind is the wrapped object:

```
160 procedure XCompress(azipfolder, azipfile: string);
161 begin
162   with TShellZip.create do begin
163     zipfile:= azipfile;
164     ZipFolder(azipfolder);
165     Free;
```

```

166 end;
167 //compress
168 end;

```

There are 3 main operators that use this object of the service routine XCompress:

1. **Constructor** (creates the object TShellZip which returns a reference.
2. **Property:** zipfile, which sets the filename for another zipfile within a string
3. **Method:** ZipFolder(), which passes the folder to compress.

But what happens behind the method ZipFolder?

We call or invoke a shell object! Another way to use dispatch interfaces is to assign the

```

122 shellobj := CreateOleObject('Shell.Application');

```

to a Variant. By assigning the interface returned by CreateOleObject to a Variant, you can take advantage of the Variant type's built-in support for interfaces. Simply call the methods of the interface, and the Variant automatically handles all IDispatch calls, fetching the dispatch ID and invoking the appropriate method.

```

V: Variant; //schema
begin
V:= CreateOleObject("TheServerObject");
V.MethodName; { calls the specified method }

```


As you already know an advantage of using Variants is that you do not need to import the type library, because Variants use only the standard IDispatch methods to call the server. The trade-off is that Variants are slower, because they use dynamic binding at runtime.

```

69 Try
70     Result:= shellObj.Namespace(ax);
71 Except

```

The Windows Shell provides a powerful set of automation objects that enable you to program the Shell with maXbox and scripting languages such as PascalScript or JScript (compatible with ECMA 262 language specification). You can use these objects to access many of the Shell's features and dialog boxes. For example, you can access the file system, launch programs, compress or decompress like in our case and change system settings.

 You can also instantiate many of the Shell objects with late binding, as we do. You can call this also scriptable Shell objects with late binding.

Many of the Shell objects became available in version 4.71 of the Shell. Others are available in version 5.00 and later. Version 5.00 became available with Win2000. The following table lists each Shell object under the version of the Shell in which the object became available.

Version 4.71	Version 5.00
Folder	DIDiskQuotaUser
FolderItemVerb	DiskQuotaControl
FolderItemVerbs	Folder2

<u>Shell</u>	<u>FolderItem</u>
<u>ShellFolderView</u>	<u>FolderItems</u>
<u>ShellUIHelper</u>	<u>FolderItems2</u>
<u>ShellWindows</u>	<u>IShellDispatch2</u>
<u>WebViewFolderContents</u>	<u>IShellLinkDual2</u>
	<u>ShellFolderItem</u>
	<u>ShellLinkObject</u>

The Shell object represents the objects in the Shell. You can use the methods exposed by the Shell object to:

- Open, explore, and browse for folders.
- Minimize, restore, cascade, or tile open windows.
- Launch Control Panel applications.
- Display system dialog boxes.

Users are perhaps most familiar with the commands they access from the Start menu and the taskbar's shortcut menu. The taskbar's shortcut menu appears when users right-click the taskbar. The following HTML Application (HTA) produces a start page with buttons that implement many of the Shell object's methods. Some of these methods implement features on the Start menu and the taskbar's shortcut menu.

<http://msdn.microsoft.com/en-us/library/windows/desktop/bb776890%28v=vs.85%29.aspx>

Next a few examples to see the use of OleObject:

{The following code shows an example of how to create an Ole Object and how to perform specific operations.}

http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/ComObj_CreateOleObject.html

```

procedure TForm1.ButtonClick(Sender: TObject);
var
    WordApp, NewDoc: Variant;
begin
    { Creates a Microsoft Word application. }
    WordApp := CreateOleObject('Word.Application');
    { Creates a new Microsoft Word document. }
    NewDoc := WordApp.Documents.Add;
    { Inserts the text 'Hello World!' in the document. }
    WordApp.Selection.TypeText('Hello World!');
    { Saves the document on the disk. }
    NewDoc.SaveAs('my_new_document.doc');
    { Closes Microsoft Word. }
    WordApp.Quit;
    { Releases the interface by assigning the Unassigned constant to the
    Variant variables. }
    NewDoc := Unassigned;
    WordApp := Unassigned;
end;

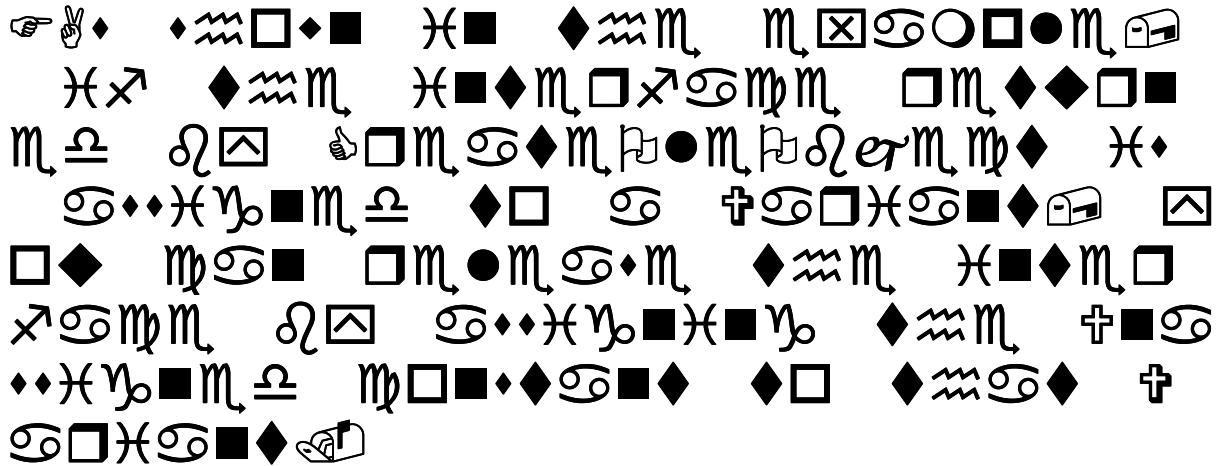
```



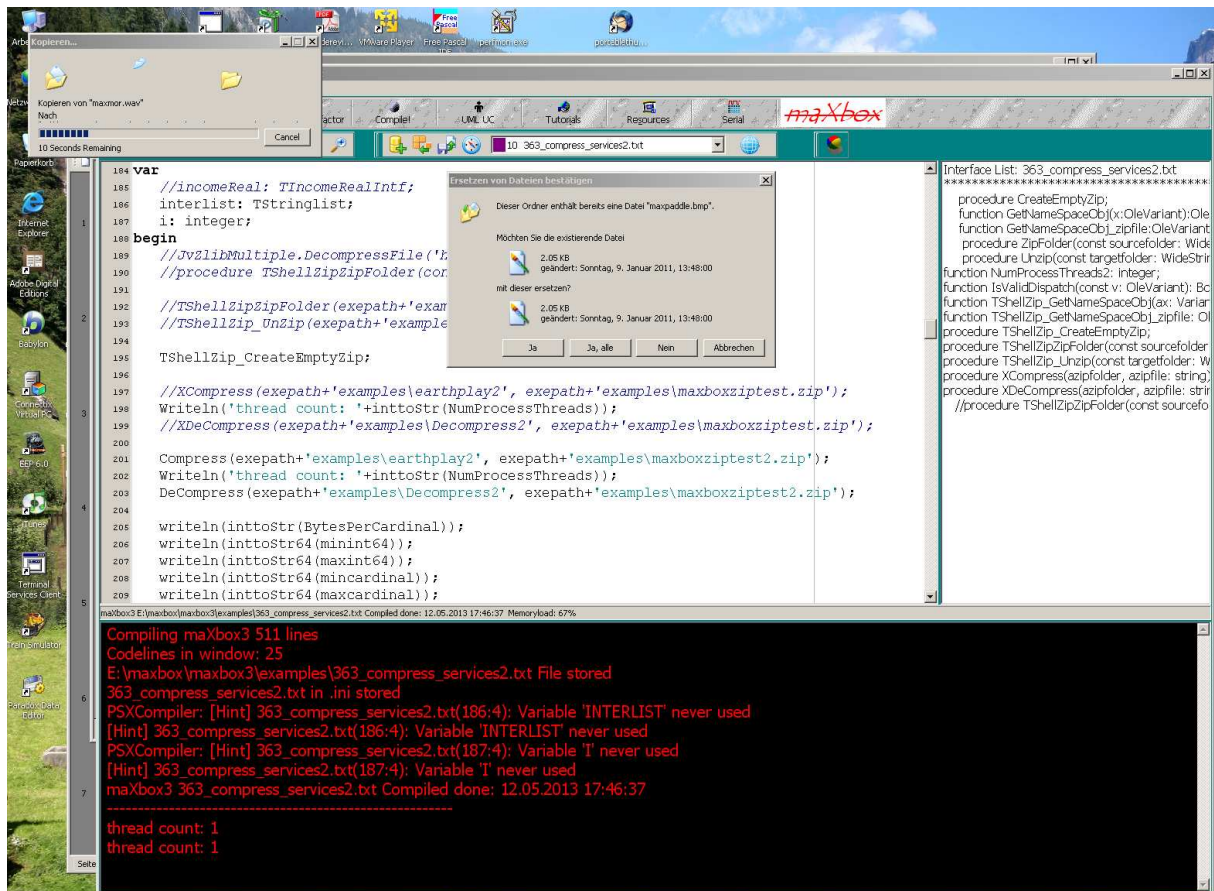
As shown in the example, you can release the interface stored in the returned Variant by assigning the Unassigned constant to that Variant. As shown in the example, if the interface returned by CreateOleObject is assigned to a Variant, you can release the interface by assigning the Unassigned constant to that Variant.

You can also use the OBJECT element to instantiate Shell objects on an HTML page. To do this, set the OBJECT element's ID attribute to the variable name you will use in your scripts, and identify the object using its registered number (CLASSID).

Now comes the fun part of our compressed file or folder, this is how it looks like to set the HEX as ASCII Windings:



Windings are a series of dingbat fonts which render letters as a variety of symbols. They were originally developed in 1990 by Microsoft by combining glyphs from Lucida Icons, Arrows, and Stars licensed from Charles Bigelow and Kris Holmes.



2: Decompress step by step in compare

Clients are seen in picture 2 as the two classes do the tasks. After creating the shell object we set the options or filter of the folder.

These are the most important options you can specify:

The `Folder` object represents a Shell folder. You can use the methods exposed by the `Folder` object to:

- Get information about a folder.
- Create subfolders.
- Copy and move file objects into the folder.

The `FolderItem` object represents an item in a Shell folder. Its properties enable you to retrieve information about the item. You can use the methods exposed by this object to run an item's verbs, or to retrieve information about an item's `FolderItemVerbs` object.

There are plenty more shell script expression tricks and operations, which can be found in Programming, or, for example a really simple one at the end:



How can we show the run dialog?

```
procedure TForm1_FormCreateShowRunDialog;

var
  ShellApplication: Variant;

begin
  ShellApplication:= CreateOleObject('Shell.Application');
  ShellApplication.FileRun;
end;
```

Conclusion: `CreateOleObject` returns a reference to the interface that can be used to communicate with the object. For `CreateOleObject` this interface is of type `IDispatch`. To create a COM object that does not support an `IDispatch` interface, use `CreateComObject`.



~~maxbox~~



Time of the day: Study more about the theory of Data Compression:

<http://www.data-compression.com/theory.shtml>

Claude E. Shannon formulated the theory of data compression. Shannon established that there is a fundamental limit to lossless data compression. This limit, called the entropy rate, is denoted by H. For example:

The implicit unpredictability or randomness of a probability p can be measured by the extent to which it is possible to compress data like a tar or zip archive:

much compressible -> less random -> more predictable

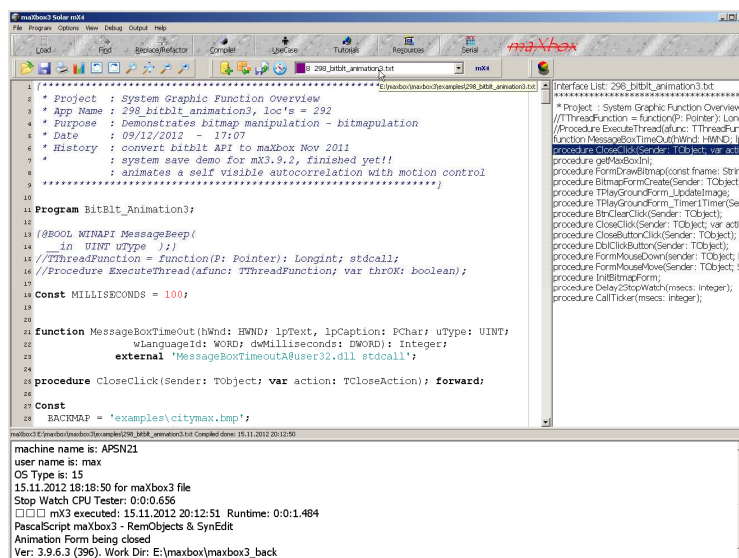
This is the entropy, a measure of how much randomness (unpredictable) it contains.

To explain entropy with a coin goes like this: E.g. a fair coin has two outcomes, each with a probability of 1/2 so the entropy is

$$1/2 \log 2 + 1/2 \log 2 = 1$$

```
PrintF('max. Entropy of coin 0.5p: %f',[0.5*log2(2)+0.5*log2(2)]);
> max. Entropy of coin 0.5p: 1.00
```

$$H = \sum_{i=1}^n p_i \log_2 p_i \text{ bits/character.}$$



Feedback @
max@kleiner.com

Literature:
 Kleiner et al., Patterns konkret, 2003, Software & Support

Links of maXbox and Data Compression:

<http://www.softwareschule.ch/maxbox.htm>

<http://msdn.microsoft.com/en-us/library/windows/desktop/bb776890%28v=vs.85%29.aspx>

<http://sourceforge.net/projects/maxbox>

<http://sourceforge.net/apps/mediawiki/maxbox/>

<http://sourceforge.net/projects/delphiwebstart>

1.3 Appendix

EXAMPLE: ProgID of Shell Objects

The ProgID for each of the Shell objects is shown in the following table.

Object	ProgID
<u>DIIDiskQuotaUser</u>	Microsoft.DiskQuota.1
<u>DiskQuotaControl</u>	Cannot late bind
<u>Folder</u>	shell.Shell_Application.Namespace("...")
<u>Folder2</u>	shell.Shell_Application.Namespace("...")
<u>FolderItem</u>	shell.Shell_Application.Namespace("...").Self or Folder.Items.Item or Folder.ParseName
<u>FolderItems</u>	Folder.Items
<u>FolderItems2</u>	Folder.Items
<u>FolderItemVerb</u>	Shell.Namespace("...").Self.Verbs.Item()
<u>FolderItemVerbs</u>	FolderItem.Verbs or Shell.Namespace("...").Self.Verbs
<u>IShellDispatch2</u>	shell.Shell_Application
<u>IShellLinkDual2</u>	Shell.Namespace("...").Self.GetLink or Shell.Namespace("...").Items().GetLink
<u>Shell</u>	shell.Shell_Application
<u>ShellFolderItem</u>	Shell.Namespace("...").Self or Shell.Namespace("...").Items()
<u>ShellFolderView</u>	Cannot late bind
<u>ShellFolderViewOC</u>	Cannot late bind
<u>ShellLinkObject</u>	Shell.Namespace("...").Self.GetLink or Shell.Namespace("...").Items().GetLink
<u>ShellUIHelper</u>	Cannot late bind
<u>ShellWindows</u>	shell.Shell_Windows or ShellWindows._NewEnum
<u>WebViewFolderContents</u>	Cannot late bind