Analytica® Decision Engine User Guide

Release 4.0

Beta—April 21, 2007

Chapter I

Lumina Decision Systems, Inc.
26010 Highland Way
Los Gatos, CA 95033
Phone: (650) 212-1212
Fax: (650) 240-2230
Web Site: www.lumina.com

Copyright Notice

Lumina Decision Systems, Inc.
26010 Highland Way, Los Gatos, CA 95033
Tel: (650) 212-1212, Fax: (650) 240-2230
Internet: support@lumina.com
http://www.lumina.com

Acknowledgements

The ADE User Guide was written by Richard Sonnenblick, Hugh Silin, Lonnie Chrisman, Max Henrion, and Richard Morgan.

Contents

Contents

Chapter 1 *Introduction*

Introduction

What is the Analytica Decision Engine?

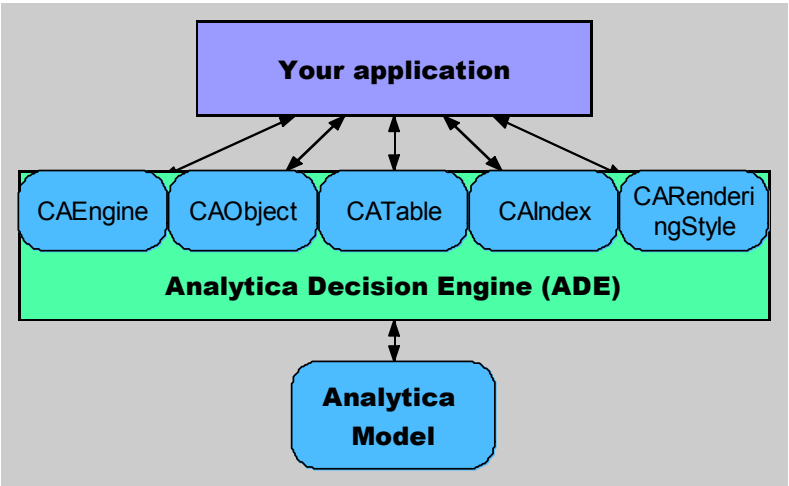
The *Analytica Decision Engine* (ADE) is a powerful COM component that helps you to programmatically access Analytica models. ADE lets you run any Analytica model on a server computer. It provides an Application Programming Interface (API) through which other application programs can create, read, check, parse, evaluate, modify, and save Analytica models. For example, you can create a user-interface accessible via a web-browser so that users can run Analytica models as web applications. Or you can use ADE to access your Analytica model from another application that may supply inputs, run the model, and collect and display results.

Although you can use ADE to build and edit models with commands issued via the API, it is usually much more convenient to use Analytica Enterprise for this purpose (see the *Analytica Tutorial* and *Analytica User Guide* for details, including the “Analytica Enterprise” chapter of the *Analytica User Guide*). Once you have an Analytica model, you can use ADE to build a custom user interface via a Web browser or other application, to interface the model with another application.

ADE is provided in two forms: An ActiveX in-process automation server. **adew.dll**. and a COM local automation server, **`ADE.exe`**, so that it is compatible with a wide range of applications. The classes, methods, and properties exposed by these servers are accessible from any programming environment that supports the use of COM, ActiveX Automation or .NET interfaces. Such environments include VB, VB.NET, ASP, ASP.NET, C#, Visual C/C++, J#, VB Script, and JavaScript. For example, you can use Visual Basic or C# to create graphical user interfaces (GUIs) on 32-bit Microsoft Windows platforms for your Analytica models, tailored to specific applications and specific classes of end-users.

Figure 1 shows a conceptual model of ADE. Your application makes calls to the functions exposed by the interface classes of ADE. Those functions then return information to your application. Server objects allow you to read, check, parse, evaluate, modify, and save Analytica models from within your applications.

Figure 1: Conceptual model of Analytica Decision Engine



Using the ADE Server

ADE provides objects of five OLE classes: **CAEngine**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle** ('CA' stands for Class Analytica). You use these classes to interact with your Analytica model through ADE. The **CAEngine** class contains methods and properties to open and close existing models, create new models, and access objects in your Analytica model.

It is important to distinguish these OLE object classes in ADE from the Analytica object classes. Analytica classes include Chance, Decision, Index, Objective, and Variable (which we refer to collectively as Variable Classes); Model, Module, and Library (which we refer to collectively as Module Classes); Functions, and Attributes. You can access Analytica objects as instances of the **CAObject** class, This class provides Properties and methods to get and set attributes of Analytica objects, including Identifier, Title, Description, and (for variables) Definition, and Value.

You can access the value of a Variable via the **ResultTable** property of class **CATable**. A **CATable** represents an Analytica Array (also known as a Table) so that you can get or set its individual elements (also known as cells). Each element may be a *number* or a *string* value (termed a *text* value in Analytica).

A **CATable** has zero or more dimensions. Zero dimensions means it is a *atomic* (it has a single element). Each dimension is identified by an Analytica Index, represented by the **CAIndex** class. A **CAIndex** has a name, and a list of labels, numbers or string, used to identify the rows or columns (more generally, *slices*) of the array. In Analytica you identify dimensions of an Array by name not by order.

The **CARenderingStyle** class provides control over formatting of returned values as numbers or text.

How to use this document

The rest of this guide has five sections:

- Installation

  This chapter explains the steps required to install the Analytica Decision Engine 4.0 on your Windows NT 4 (>SP 6), 2000, XP, or Vista computer.

- Tutorial

  This chapter shows you how to use the *Analytica® Decision Engine* (ADE) from within a Visual Basic program, and steps you through building your first ADE application using Visual Basic.

- Using the Analytica Decision Engine Server

  This section provides a step-by-step guide to the functionality accessible through ADE. You should read this section to get better acquainted with the classes, and their methods and properties. By using the sample code fragments presented in this section in your code, you can begin accessing information in your models from your Visual Basic applications immediately.

- Working with Analytica Models, Modules, and Files

  This chapter contains examples of various common operations and manipulations you might perform on objects in your Analytica model.

- Analytica Decision Engine Server Class Reference

This chapter provides reference materials on the four object classes in ADE and their properties and methods. Information that can be found in this chapter includes method syntax, data types, and property access information. Refer to the information in this section after you've read through the section, "Using the Analytica Decision Engine Server", and have specific questions about particular methods and properties.

Chapter 2 *Installation*

This chapter explains the system requirements for the Analytica Decision Engine (ADE). It describes how to install, upgrade, and uninstall, the Analytica Decision Engine.

Installation

System Requirements

- Windows NT 4.0 (>SP6), 2000, XP, Windows Server 2003, and Vista.
- 20 MB of hard drive space (you will need more space to develop your applications).
- 256 MB of RAM
- You will also need a development language environment to build your application using ADE. This could be Visual Studio with VB.NET, C#, ASP.NET, VC++, or any other COM or .NET-enabled development environment. You can also use ADE from Microsoft Office Visual Basic for Applications (VBA) or from Windows Scripting Host (CScript.exe or WScript.exe), as well as other COM-enabled or ActiveX automation-enabled packages.

You can obtain all the files for installation of ADE from the ADE CD-ROM or you can download the installer from

http://www.lumina.com/ana/support/download.htm

The installation contains the ADE in-process automation server (**`adew.dll`**), the ADE local automation server (**`ADE.exe`**), auxiliary files needed by ADE, this *ADE User Guide*, and example programs.

Installing the Analytica Decision Engine files

Installing from the network

Obtain an ADE 4.0 license code from Lumina. This will be supplied to you, usually through e-mail, when you purchase ADE. You must complete the installation within three days after the license code is issued to you.

Download the ADE setup executable. The location of the file will be provided to you when you receive your license code. Save the file to disk.

Run (e.g., double click on) the file just download to begin the ADE installer.

Note: *Before running the ADE installer, you must have the Windows System Installer (WSI) already installed on your system. This is guaranteed to already be on your system if you are using Windows XP, or if you have previously installed Analytica. If you have installed any recent Windows software, it is almost certainly present. If it is not present, the easiest remedy is to install Analytica or later.*

Follow the instructions. Read and agree to the license agreement, select a directory for the installation, and enter your license code when prompted.

If the installer reports that your license code is stale, go to ***`http://lumina.com/ADE/staleLicense`*** and obtain a fresh code. After you obtain a fresh code, be sure to enter the license code within three days.

Installing from CD

Insert the Analytica Decision Engine CD into your CD-ROM.

If the installer does not automatically start, run the setup.exe program on the CD-ROM.

During the setup, you will need to select a directory for installation, to read and agree to the licensing terms, and to enter the license code supplied to you by Lumina Decision Systems when you acquired ADE.

If the installer reports that your license code is stale, go to http://lumina.com/ADE/staleLicense and obtain a fresh code. After you obtain a fresh code, be sure to enter the license code within three days.

After following the above steps, the following files should exist in the directory in which you installed ADE:

- **`Adew.dll`**
- **`ADE.exe`**
- **`Analytica.ini`**
- **`Analytica.i`**
- **`ODBC4Analytica.dll`**
- **`license.txt`**
- **`SolverSDK.dll`**

Two ADE manuals are installed into a subdirectory called docs. These are:

- **`ADE User Guide.pdf`** (this document)
- **`ADE Scripting.pdf`**

Four example programs should also have been installed in that directory underneath the examples directory. They are as follows:

`Tutorial`—the program referred to by the Analytica Decision Engine Tutorial. It is recommended that you read the Analytica Decision Engine Tutorial completely before writing your own programs that depend on ADE.

`AdeTest`—a program that allows you to call or test the methods of ADE objects through a GUI. You can run AdeTest.Exe (in the bin directory) directory, or you can trace through the code in the Visual Studio.NET 2005 debugger to observe each method being called.

`asp_exam`—a program that shows how to access ADE through a Microsoft ASP program.

`excel_exam`—a program that shows how to access ADE from any application with Visual Basic for Applications (VBA) support, including the Microsoft Office suite of applications.

Entering a new license code

If you have previously installed an earlier version of ADE and need to enter a new (different) license code, follow these steps:

Open a command prompt.

Select *Start* → *Run* and type **`Cmd.exe`**.

Change directory (using the **`cd`** command) to the directory where you installed the earlier version of ADE.

Type: `ADE /RegServer`

A dialog will appear prompting you enter your new license code.

Upgrading from an earlier version of ADE

ADE 4.0 has been configured to install without disturbing previously installed versions of ADE. This allows you to compare the performance and output of your application under the different versions. However, it also means that your existing applications will continue to use the previous version until you have changed them to use ADE 4.0. The section "Migrating to ADE 4.0" describes the changes. When you have completed your migration to ADE 4.0, you can uninstall the previous version of ADE.

If you are installing ADE 4.0 on a computer that has previous contained a beta release of ADE 4.0, the beta version will be upgraded (replaced) by the installer, but after the installer completes, you will need to follow the steps in the previous section, *Entering a new license code,* in order to enter your non-beta license code.

Uninstalling ADE

To uninstall ADE, select **Add/Remove Programs** in the Windows Control Panel. Scroll to find ADE 4.0. Press the **Change/Remove** or **Add/Remove…** button, depending on your operating system, and select **Remove**.

The uninstall will only remove files that were placed on your system by the installer. If you have compiled some of the examples, there may be some files and directories, created during those exercises, which are not removed. To remove these as well, find the install directory (*e.g.,* `C:\Program Files\Lumina\ADE 4.0`) and remove them after the uninstall has been run.

Chapter 3 *The Analytica® Decision Engine Tutorial*

This Tutorial shows you how to use the *Analytica® Decision Engi*ne (ADE) from within a Visual Basic program.

The ADE Tutorial

Your First ADE Application

First let us write a simple ADE application from scratch, just to be sure that everything is set up correctly. Follow these steps:

1. Bring up Visual Studio.NET
2. Select New Project, Project Type "Visual Basic Projects", and template "Console Application". Select a project name, e.g., "FirstADEtry" and an appropriate folder location.
3. From the Project menu, select “Add Reference” and select the “COM” tab in the dialog. Find and select “Analytica Decision Engine Local Server 4.0” (**Ade.exe**) and click [OK]. (If you cannot find this entry in the list of COM servers, then ADE 4.0 is not properly installed. See “Installation” on page 5 for instruction on how to install ADE before reading further.)
4. Add to the **Module1** class as follows:

```
Imports ADE
Module Module1
    Public m_ade As CAEngine
    Sub Main()
    Dim filename, modelname As String
    filename = "C:\Program Files\Lumina\Analytica 4.0\Tutorial
                Models\Car Cost.ana"
        m_ade = New CAEngine
        modelname = m_ade.OpenModel(filename)
        If modelname = "" Then
            Console.Write(filename & " not found")
        Else
            Console.Write("Congratulations on opening" &
                          modelname)
        End If
    End Sub
End Module
```

5. Now run the program. If your program prints “Congratulations on opening Carcosts”, you have just successfully written your first ADE program.

This first program did the following:

- created a **CAEngine** automation object called **m_ade** (using `new CAEngine`),
- opened an Analytica model (using the **OpenModel** method of **CAEngine**), and
- displayed the name of the model (the return value of **OpenModel**).

We will go into the details of these functions, and many more functions in the next section.

What’s next?

We will not attempt to explain *all* of the features of ADE in this tutorial. Those are described in the following chapters of this guide. Here, we will give you the background to explore the more advanced features of ADE on your own.

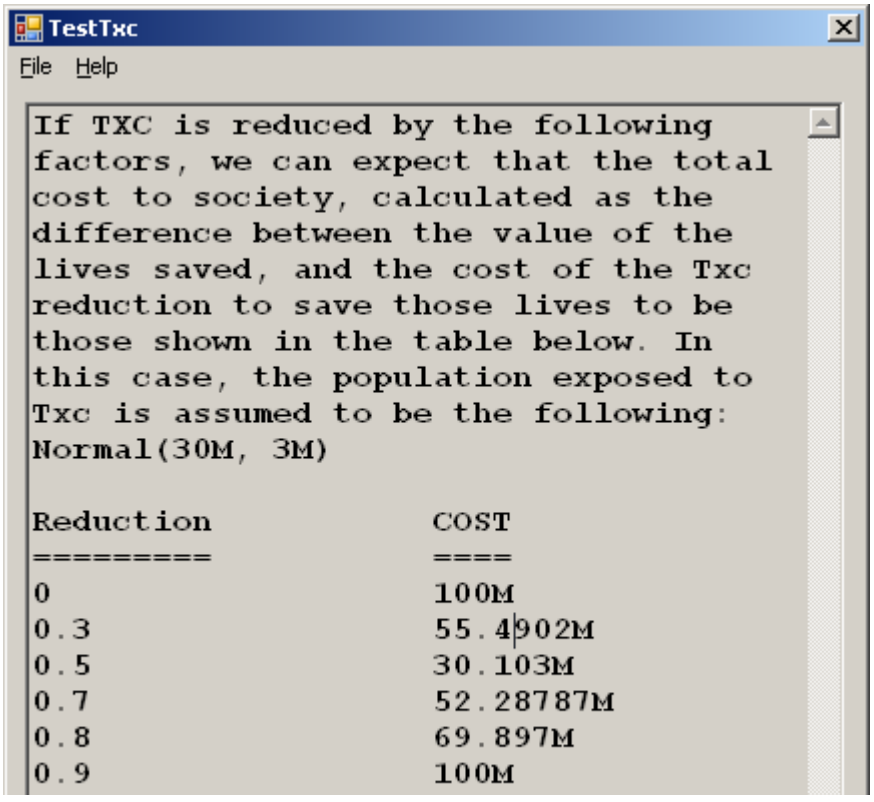
From this point, we will use the example model called `Txc.ana`. You can find `Txc.ana` under the **Risk Analysis** folder under the **Example Models** folder installed with Analytica. If you cannot find it, or if you opted not to install the examples when you originally installed Analytica, there is a copy in the **`Examples\Tutorial`** folder in the directory where you installed ADE.

The Txc model demonstrates risk-benefit analysis of reducing the emissions of fictitious air pollutant "TXC". Please open the Txc model with Analytica in order to see how it works.

The example Visual Basic.NET program called **TestTxc** under your **Ade Examples\Tutorial.NET** folder shows many aspects of ADE: It creates an ADE automation object, opens the **Txc.ana** model with this object, gets the definition of the *“Population Exposed”* variable, evaluates the *“Total Cost”* variable, prints out the result of the *“Total cost”* variable as a table by getting at the individual components of the table, and changes the definition of the *“Population Exposed”* variable. It then gets the result of the *“Total cost”* variable again, to see what effect the change of definition for *“Population Exposed”* had on the *“Total Cost”* variable. If things are set up properly, **TestTxc** displays the window show in Figure 1: “Text Txc Window”.

The application displays the Definition of the *“Population Exposed”* variable ("Normal (30M, 3M)"), and the table associated with *“Total Cost”*, based on the definition of “Population Exposed”. You can change the definition of *“Population Exposed”* by selecting **File** > **Change Population Exposed** from the main menu and seeing the effect this has on the *“Total Cost”* table.

Figure 1: Text Txc Window



Distinguishing Title from Identifier

Whenever an ADE function requires a variable, you must pass it the *identifier* of the variable, not its *title*. This can be confusing since Analytica normally displays the titles of each variable in an influence diagram. By default, when you first create each object, Analytica automatically creates an identifier based on the title. It substitutes underscore "_" for each blank or other character in the Title that is not a letter or number.

You can show the identifiers in an influence diagram by pressing *control-y* (or by selecting **Show by identifier** from the **Object** menu). For model `Txc.ana`, you can see that the identifier of the variable titled *"Population Exposed"* is "Pop_exp". It is important to use "Pop_exp" as the identifier when passing this variable to ADE functions. ADE would not be able to find the variable if you pass *"Population Exposed"* instead, and would return an error.

Creating an ADE Object from within Visual Basic

If you haven't already, load the project called `Examples\Tutorial\TestTxc.sln` into Visual Basic.NET, and view the code for the file called `TestTxc.vb`. It looks like this:

```
Imports ADEW
. . .
Public adeEngine As CAEngine

Public Sub Main()
        Dim exeDirectory, theModel As String
        Dim theModelString As String
        exeDirectory = VB6.GetPath
        theModel = exeDirectory & "\" & "Txc.ana"
adeEngine = New CAEngine
...
        theModelString = adeEngine.OpenModel(theModel)
        ...
        frmMain.DefInstance.Show()
End Sub
```

At the very top of the file, it declares the automation object **adeEngine** as a **CAEngine** object. Via this object, we can access all of the public functions exposed by **CAEngine** (see "ADE Server Class Reference" *on* page 53 for a complete listing). This line will then create the **CAEngine** Object:

```
adeEngine = New CAEngine
```

The **adeEngine** variable now holds our in-process **CAEngine** object.

If we want to use the local (out-of-process) server version of ADE, we can add a reference to the project to the "Analytica Decision Engine Server 4.0" COM component and change the top line from "Imports ADEW" to "Imports ADE".

Here is another way to obtain a new **CAEngine** object, which does not require adding a reference to the project:

```
adeEngine = CreateObject("ADEW4.0.CAEngine") ' in-process
adeEngine = CreateObject("ADE4.0.CAEngine")  ' out-of-process
```

To understand the pros and cons of using an in-process server versus as out-of process (or local) server, and which automation server to use for different scenarios, see “In-Process vs. Out-of-Process” on page 22, as well as other books related to COM servers.

COM vs. Automation Interface

In the above example, we used a COM interface to call ADE. In a COM interface, the object (**CAEngine** in this case) is declared as **CAEngine**, and the compiler resolves each member function and can detect several obvious errors at compile time. In addition, Visual Studio can provide a list of methods and parameter types as tool tips as you program, which is helpful when writing programs that use ADE. COM calls are slightly faster than Automation calls, but the speed difference is not usually significant in applications of ADE. With ADE 4.0, we recommend using the COM interface if your programming language supports it.

In VB Automation, you can declare an object simply as **Object**, rather than a more specific types such as **CAEngine, CAObject,** *etc*. When ADE methods are called using Automation, the methods are resolved at run-time. At compile time, the compiler does not know whether your m_ade object has a method named OpenModel. In VB, the syntax for calling a COM method or an Automation method is identical — the only difference is whether the object’s type is declared explicitly.

In VC++ and C#, the syntax for calling COM is not the same as for Automation. In these cases, COM is much more convenient, while Automation can get rather tedious. However, some languages, including VBScript and other scripting languages, support only Automation and not COM.

Opening a Model with ADE

We will now open the **Txc.ana** model, and show the main window of our application. Use the following call:

```
theModelString = adeEngine.OpenModel(theModel)
frmMain.DefInstance.Show
```

The OpenModel function of **CAEngine** will open the model. If successful, the variable **theModelString** will contain the name of the model. Otherwise, it contains an empty string. Although we haven’t done so in this example for the sake of brevity, you should check to see that the string returned from **OpenModel** isn’t empty. If it is, there was an error in opening your model. You can find out what kind of error with the **ErrorCode** and **ErrorText** properties of **CAEngine** (**adeEngine.ErrorCode** and **adeEngine.ErrorText**). We will see how to use these two properties later on. For a listing of all the error codes, see Appendix A: “Error Codes” on page 76.

Retrieving Objects from the Analytica Model

The next step is to retrieve objects (Variables, Modules, Functions, etc.) from our model, so that we can access their attributes (Definition, Title, Class, etc.). Our example model (Txc.ana) manipulates the **Pop_exp** and **Cost** objects. In particular, it modifies **Pop_exp** to see how this effects the **Cost** object.

The **PrintAttributes** function in the file **frmMain.frm** of our **TxcTest.vbproj** (TxcText.sln) project shows how to do this. This function is first called by the **Form_Load** function of **frmMain.frm,** when the application starts, to display the **Cost** table. It is also called whenever we wish to print out the current result of our **Cost** table. The function looks as follows:

```
Public Sub PrintAttributes(ByRef inputIdentifier As String, ByRef
     outputIdentifier As String)
        Dim inputObject, outputObject As CAbject
        Dim resultTable As CATable
        Dim definitionAttrInput As String
        inputObject = adeEngine.GetObjectByName(inputIdentifier)
        outputObject = adeEngine.GetObjectByName(outputIdentifier)
        definitionAttrInput = inputObject.GetAttribute("definition")
        resultTable = outputObject.resultTable
        Call PrintResultTable(resultTable, inputIdentifier,
                              definitionAttrInput, outputIdentifier)
End Sub
```

PrintAttributes gets with the variable identifiers, **Pop_exp** passed as parameter **inputIdentifier** and **Cost** passed as parameter **outputIdentifier**. It fetches the corresponding objects using the **GetObjectByName** function of **CAEngine** thus:

```
inputObject = adeEngine.GetObjectByName(inputIdentifier)
outputObject = adeEngine.GetObjectByName(outputIdentifier)
```

If **GetObjectByName** succeeds, it returns an object of type **CAObject**. You then use the functions of **CAObject**. See “SendCommand(command)” on page page 59 for a listing all **CAObject** functions. If **GetObjectByName** fails, the return value is **`Nothing`**. The code should check to make sure that the result from **GetObjectByName** is valid. If not, use the **ErrorCode** and **ErrorText** properties of **CAEngine** to get more information about the error. For example:

```
Set inputObject = adeEngine.GetObjectByName(inputIdentifier)
If inputObject Is Nothing Then
     MsgBox("This error from GetObjectByName occurred: "& _
           vbCrLf & adeEngine.errorCode & ":" & adeEngine.errorText)

Else
     'inputObject valid
End If
```

Getting Object Attributes

Each Analytica object has a set of Attributes (analogous to “Properties”), such as Identifier, Title, Description, and Class. You can use the **GetAttribute** function to obtain an Attribute from an Analytica Object. For example, to get the Definition of **inputObject** (currently, the cost):

```
definitionAttrInput = inputObject.GetAttribute("definition")
```

In the **Txc.ana** model, the definition of **Pop_exp** is “Normal (30M, 3M)” which we store in **definitionAttrInput**.

Evaluating Objects and Retrieving Results

Use **Result** or **ResultTable** methods of **CAObject** to get the value of a variable. ADE will automatically evaluate the variable first if necessary. Use the **Result** method if you are sure the result will be ***atomic***, i.e. a single element. Otherwise, use **ResultTable**, which retrieves the result as an *array*. An atomic result is treated as a special case of an array, one with zero dimensions. If the value is atomic, the method **AtomicValue** returns its single value as a number or string.

By default, **Result** and **ResultTable** return the *Mid* value of the result — i.e. the result of ADE evaluating it as deterministic. For a probabilistic value, set the **ResultType** property of **CAObject** to the desired uncertainty view — Mean, Sample, PDF, CDF, Confidence bands, or Statistics,. (See “ResultType” on page 60 for details.) We get the value of **outputObject**, thus:

```
resultTable = outputObject.ResultTable
```

The result is a **CATable** object, which lets us accesss individual elements in a table.

If you call **Result** to get an array (or table) value, it returns the array as a string, listing the indexes and elements separated by commas. It is usually easier to use **ResultTable**, so that you don’t have to parse elements of the table from the string.

Getting the Index Elements of a Table

An Analytica table has zero or more indexes. If it has one index *e*, then it is one-dimensional; if it has two indexes, it is two-dimensional, and so on. A zero-dimensional table holds a single *atomic* (or *scalar*) value. You can use the **NumDims** function of **CATable** to get the number of dimensions (same as number of Indexes) of a table. To get at the individual indexes of a table, use methods **IndexNames** and **GetIndexObject** of **CATable**.

The function **PrintResultTable** in frmMain.frm shows the use of these two functions . **PrintResultTable** is called from **PrintAttributes**, and does the actual work of printing the table that shows up in our **TestTxc** application. (For brevity, we show only the parts of this function related to ADE).

```
Public Sub PrintResultTable(ByRef resultTable As CATable,
                            ByRef inputIdentifier As String,
                             ByRef definitionAttrInput As String,
                             ByRef outputIdentifier As String)
        Dim theIndexName, theTableName As String
        Dim theIndexElement As String
        Dim theTableElement
        Dim theIndexObj As CAIndex
        Dim numEls As Integer
        Dim spaces, i As Integer
        Dim lenStr As Short
        Dim OutputStr As Short
        Dim spaceString, underlineString As String
        ...
        theIndexName = resultTable.IndexNames(1)
        theTableName = resultTable.Name
        theIndexObj = resultTable.GetIndexObject(theIndexName)
        numEls = theIndexObj.IndexElements
        For i = 1 To numEls
            theIndexElement = theIndexObj.GetValueByNumber(i)
```

```
            theTableElement = resultTable.GetDataByElements(i)
            ...
        Next i
        InformationPane.Text = outputString
    End Sub
```

The lines of **PrintResultTable** that get an index of a table are as follows:

```
theIndexName = resultTable.IndexNames(1)
theIndexObj = resultTable.GetIndexObject(theIndexName)
```

We get the name of first index using the **IndexNames** function of **CATable**. We pass it into the **GetIndexObject** function of **CATable** to get a **CAIndex** object that represents our index. This automation object returns information about its corresponding index. If this function fails, it returns `Nothing`. In that case, use **ErrorCode** and **ErrorText** functions of **CAEngine** to find out why.

Retrieving information from CATable & CAIndex objects

PrintResultTable also shows how to get information from **CATable** and **CAIndex** objects. This code gets the index and table elements of the **Cost** table:

```
numEls = theIndexObj.IndexElements
For i = 1 To numEls
    theIndexElement = theIndexObj.GetValueByNumber(i)
    theTableElement = resultTable.GetDataByElements(i)
    ...
Next i
```

The **IndexElements** property of **CAIndex** returns the number of elements in the (first) index. The **GetValueByNumber** function of **CAIndex** gets individual index elements.

To get the individual table elements of the Cost table object, **resultTable**, we use the **GetDataByElements** function of **CATable**, passing in the coordinates of the element in the table.

When we retrieve an individual element of our **CATable** object (**resultTable**), we take advantage of the fact that the table is one-dimensional. Therefore, we only need to pass **GetDataByElements** a single number representing the position in our table. If we were dealing with two or more dimensions, however, we would need to pass **GetDataByElements** an array specifying the coordinates of the element of our table to retrieve. So, if we want to retrieve the element at position (4,3) of a 2-dimensional table, we would write:

```
Dim W as Variant 'return element
Dim IndexPtrs(1 To 2) As Variant 'position in table
     ...
IndexPtrs(1) = 4
IndexPtrs(2) = 3
W = resultTable.GetDataByElements(IndexPtrs)
```

Controlling Formats of Atomic Values

Each atomic value in a **CATable** may be a number, string, or one of a few other basic types (*e.g,.* Null, Undefined, Reference, or Handle). These are returned as ***variants***, a data structure understood by Visual Basic, specifying the type and value. The **RenderingStyle** property of **CATable** controls how the underlying Analytica value is mapped to the Visual Basic variant.

For example, it can return a numeric value as a number, or a string using the Analytica model's number format setting. If it is formatted, an option controls whether to truncate the number of digits, or to return it with full precision.

In the **PrintResultTable** subroutine, located in **frmMain.vb**, the rendering style is explicitly specified:

```
resultTable.RenderingStyle.NumberAsText = True
resultTable.RenderingStyle.FullPrecision = False
resultTable.RenderingStyle.StringQuotes = 2
```

The first line specifies that numeric values should be formatted as text according to the number format associated with the result object. For example, in the program output, we see “30.103M” instead of 30102995.6639812, which would likely be displayed if we had let Visual Basic concatenate the numeric value to our result string. In the event that a string-valued cell occurs in the result, it will be returned with explicit double quotes around the value. See “Class CARenderingStyle” on page 72 for additional properties available through the **CARenderingStyle** object.

Other Ways to Access Tables

There are several ways to access the elements of a multi-dimensional **CATable**. Some may be more convenient in certain scenarios than others.

The first way is to use the **GetDataByElements** or **GetDataByLabels** methods of **CATable**, shown in above. In this case, you supply the coordinates of the cell whose atomic value you which to retrieve.

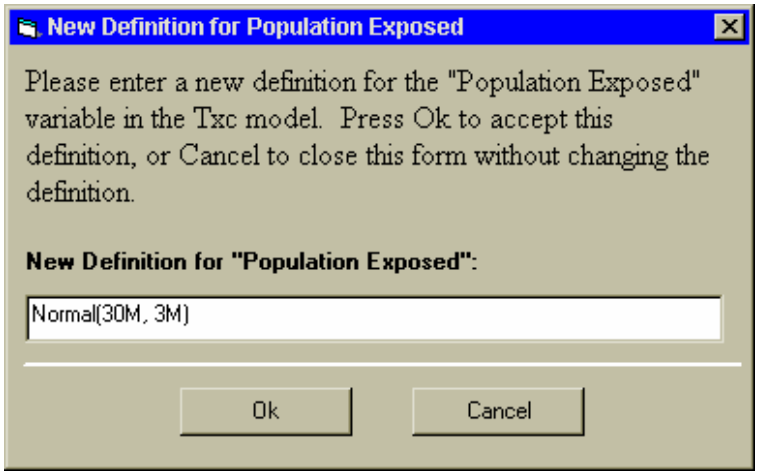
A second way is to use the **Slice** or **Subscript** methods of **CATable** to obtain a new **CATable** object having one less dimension. By repeatedly reducing the dimensionality, you will eventually reach zero dimensions, in which case you have a single atomic value. At that point, the **AtomicValue** method of **CATable** returns this value. The **AtomicValue** method is the only way to access a scalar value (since it doesn't have a coordinate). You must use this method if you need to generate a graph image of a slice of the full result.

A third way is to use the **GetSafeArray** method of **CATable**, to convert the multi-dimensional array into a Safe Array (or into a .NET array). You can then manipulate the multi-dimensional array directly in VB or other .NET language. Since there is no inherent ordering to Analytica dimensions, but Safe Arrays and .NET arrays have an explicit ordering, you must first use the **SetIndexOrder** of **CATable** to specify the ordering of dimensions before calling **GetSafeArray**. (Not necessary if you know your array to be one-dimensional,)

.Modifying Objects

A custom application often gets input from a user or other external source, to transfer into input variables in the Analytica model. You can do this either by setting the definition of an input variable, or by using a definition table.

TestTxc shows how to modify the definition of Pop_exp, which is a model input that effects the Cost result variable. To set the definition in the example, select File|Change Population Exposed from the main menu. The following dialog appears:



You can enter a new definition for into the field, and press OK. At that point, the main window of our application will display the new value of Cost. The OkButton_Click function in ChangeDef.frm is called when the Ok button is pressed in the above dialog. It modifies the definition of Pop_exp, and then calls the PrintAttributes function that prints the result of Cost.

The function looks as follows:

```
Private Sub OkButton_Click(ByVal eventSender As System.Object,
        ByVal eventArgs As System.EventArgs) Handles OkButton.Click
    Dim errorText As String
    Dim pop_exp_Object As CAObject
    Dim errorCode As Short
    Dim errorString As String
    newDefinition = PopExposedDef.Text
    pop_exp_Object = adeEngine.GetObjectByName("pop_exp")
    pop_exp_Object.SetAttribute("definition", newDefinition)
    errorCode = adeEngine.errorCode
    If errorCode <> 0 Then
        MsgBox("This error occurred while processing your
definition: " &
                vbCrLf & vbCrLf & adeEngine.errorText)
        PopExposedDef.Focus()
    Else
        Me.Close()
        frmMain.DefInstance.PrintAttributes("Pop_exp", "Cost")
    End If
End Sub
```

This function grabs the new definition typed into the New Definition for Population Exposed field and sets it to the Pop_exp object by using the SetAttribute function of **CAObject** object. It then calls **PrintAttributes**, which evaluates the Cost object, and prints the new table.

To set a new definition for the pop_exp variable, we get the **CAObject** for Pop_exp, and set it's definition to the definition typed in by the user. This is done with the following code:

```
pop_exp_Object = adeEngine.GetObjectByName("pop_exp")
pop_exp_Object.SetAttribute "definition", newDefinition
```

Whenever you call **SetAttribute**, you should check the **ErrorCode** of the **CAEngine** automation object (**adeEngine**), in case the definition is illegal.

Try entering a new definition such as Uniform(25M,35M) and press **Ok**. When the definition of **pop**_exp is changed, the result for **Cost** gets recomputed by ADE when **ResultTable** is next called for the Cost variable (when the application window is repainted).

Graphing with ADE

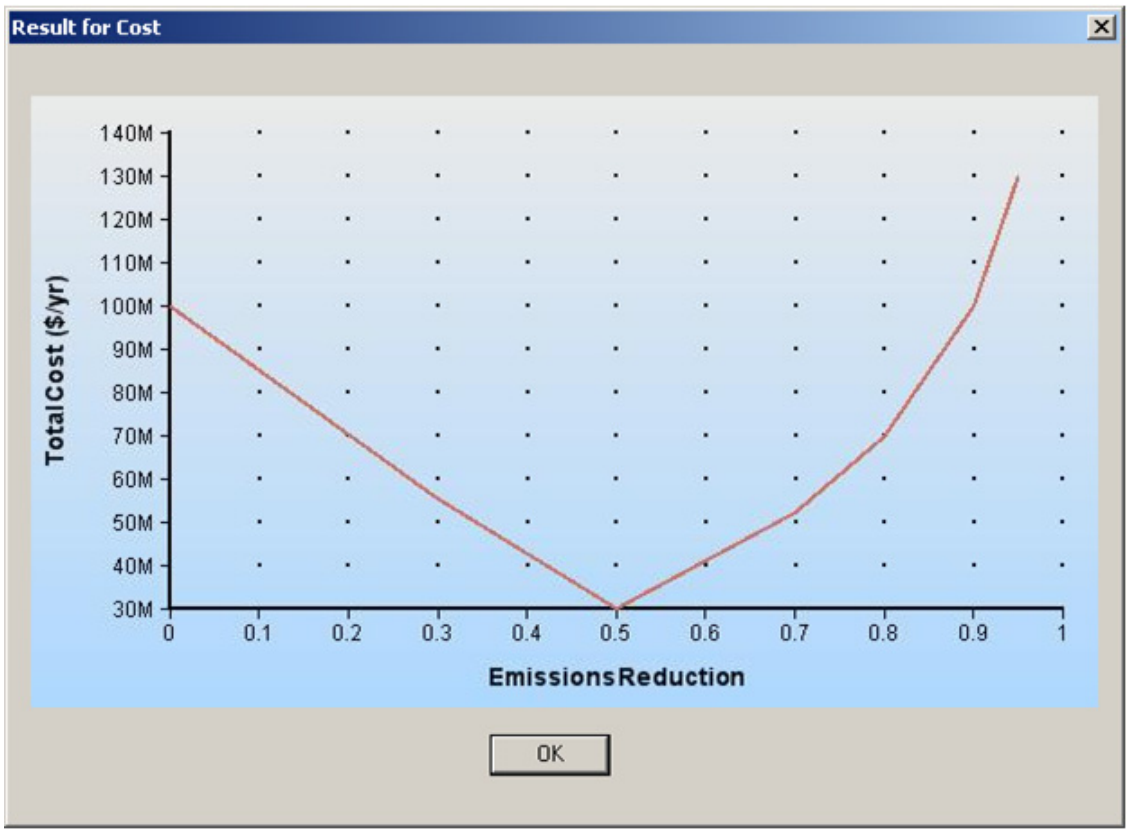
You may generate a chart or graph to display an array-valued or uncertain result, using the same graphing engine used by Analytica 4.0. In ADE 4.0 you can use the **GraphToFile** and **GraphToStream** methods of **CATable**. The graphs are returned in several possible image formats, such as “image/bmp” or “image/jpeg”.

The easiest way is to select from the many graphing options available is to open your model with Analytica Enterprise. You can experiment with the settings for the default for all variables or for the override for selected variables to see how they look. When you've chosen the settings you want, save the model. ADE will then use these settings when producing result graphs for the each variable.

For higher-dimensional results, some work may be necessary to select the slice of the result that will be plotted and the specific *pivot* (i.e., which dimensions will appear on the X-axis versus in the key). The **Subscript** or **Slice** methods of **CATable** can be used to select the particular slice to be plotted and the **SetIndexOrder** can be used to control the pivot. See the “Class CATable” on page 64 for details. In our Txc example, we have a one-dimensional result (*Cost*), and do not need to worry about slicing or pivoting.

In the example, the **GraphToStream** method is used to transfer the graph image directly from ADE to a user-interface method. **GraphToStream** is a bit more complicated to use than **GraphToFile**, since **GraphToFile** requires little more than a file name to where it writes the image. To use **GraphToStream**, we must set up a stream in memory, allow ADE to write to that stream, and then reconstitute the image from that stream. Because .NET streams are not compatible with COM streams, you need to use a **StreamConnector** class provided with ADE. The **GraphResult_Click** routine in

frmMain shows the use of **GraphToStream**. Select the **Graph Result** menu option from the main application window, and it will show:



Conclusion

In this tutorial, we introduced several important aspects of the Analytica Decision Engine. We saw how to create the ADE server object, open a model with ADE, get at an individual object in a model, evaluate objects, access elements in a table, and modify objects in a model. But, ADE can do a lot more! We hope that you have learned enough about the basics so that you can now explore the more advanced features on your own. We recommend that you now read the rest of this guide in order to learn about what else ADE can do.

Chapter 4

Using the Analytica Decision Engine Server

This chapter describes the Analytica Decision Engine server classes: **CAEngine, CAObject, CATable, CAIndex**, and **CARenderingStyle**, and the server class architecture.

Using the Analytica Decision Engine Server

ADE exposes five classes: **CAEngine, CAObject, CATable, CAIndex**, and **CARenderingStyle** ('CA' stands for Class Analytica).

- The **CAEngine** class contains methods and properties that allow you to open and close existing models, create new models, create new Analytica objects, and access Analytica objects contained in your model.
- The **CAObject** class contains methods and properties that allow you to set and obtain information about the Analytica objects (such as variables or modules) that you obtain from the **CAEngine** class.
- The **CATable** class is used to examine multi-dimensional results or to view and modify multi-dimensional definition tables (a.k.a., Edit Tables).
- A **CAIndex** object provides access to one dimension of a multi-dimensional **CATable**.
- The **CARenderingStyle**, which is new to ADE 4.0, allows you to control or alter the format in which values returned from ADE. The following sections describe how to access these Analytica Server objects from Visual Basic or C#.

Analytica Decision Engine Server Class Architecture

COM, Automation, and .NET

ADE 4.0 supports two calling conventions: COM and ActiveX Automation. COM is an early-binding convention in which the methods and data types are resolved when your application code is compiled. Automation is a late-binding convention where method calls are resolved at run time. The COM convention is somewhat more efficient, although for most applications, the difference in efficiency is far overshadowed by the time required to compute your model's results.

In Visual Basic, the syntax for calling a method using COM or Automation is identical, and which is used depends on how you declare your objects. In other languages, such as C# or C++, the method of invocation may look quite different. In C# and C++, it is generally more convenient to use the COM interface. VBScript (used by the Windows Scripting Host and older versions of IIS ASP) supports only the Automation interface.

The COM interface can be used quite transparently from a .NET environment such as Visual Studio 2005. The .NET programming environment wraps COM objects with a .NET Interop object, which gives ADE interfaces the appearance of being .NET interfaces.

In ADE 3.1 and before, the Automation interface was the recommended convention; however, with the ADE 4.0 release, we now recommend the COM interface unless this is not an option in your programming environment (such as VBScript).

In-Process vs. Out-of-Process

ADE can be launched either in-process or out-of-process. When launched in-process (ADEW), the Adew.dll library is loaded into your application's process space. When

launched out-of-process (ADE), the `ADE.exe` server is launched and runs in a different process. Both types of server use the exact same class interfaces, so the choice of which type of server to use can be changed usually by changing a single line of code – i.e., the line that instantiates the **CAEngine**.

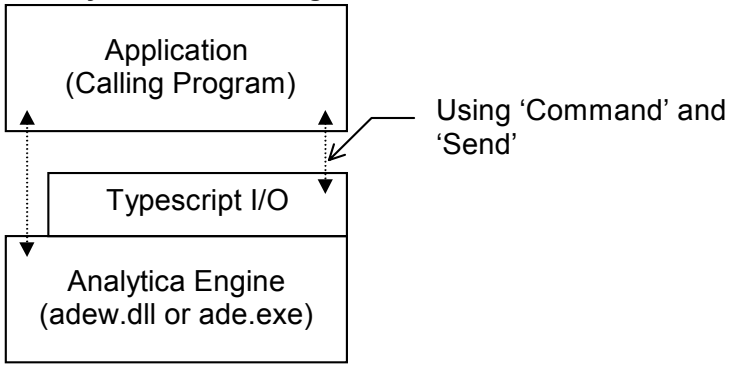
In-process servers have a slight performance advantage, but come with several restrictions. First, the *apartment* threading model of ADEW must be compatible with your application's threading model. For example, The Microsoft IIS web server (IIS 5.0 or later) will not allow you to use an apartment threaded component under its default settings. Also, you will be restricted to have only one **CAEngine** instance (and thus, only one model) in memory at any one time.

Out-of-process instances of ADE run in a different process, and can even be configured to run on a totally different computer, from your application. Because data must be "marshaled" across process boundaries, it is a bit less efficient, but it is far more flexible than the in-process. Your program can make use of multiple simultaneous instances of ADE, each with a separate model instance loaded. As such, the out-of-process is almost always preferred for web applications (e.g., you can have one ADE instance for each session).

Typescript

In addition to the Program Interface, ADE has a fully functional command interface, known as the typescript. The typescript language is described in the *Analytica Scripting Guide,* and allows access to all of ADE's functionality. The API provides a more convenient, object-oriented, set of functions for communication with the engine from Visual Basic and C++ applications. A calling program can use the API functions, or it can pass typescript commands directly to the typescript interface.

Figure 1. The Analytica Decision Engine Architecture



Security Permissions under IIS 5

When creating a web application that uses ADE from within Microsoft's Active Server Pages (ASP / ASPX) under Internet Information Server (IIS), you may need to configure permission settings in order to instantiate and access the ADE COM component from your program.

When creating a web application or web service, you should use the out-of-process ADE server. When your ASPX application is executed while serving a web page request, the ADE COM component will be launched and accessed from a special internal Window's account name. Even though your programs can create and access ADE when run under your account, the same access may not exist for ASP or ASPX

programs. To configure security permissions so that your ASPX application can use ADE, follows these steps:

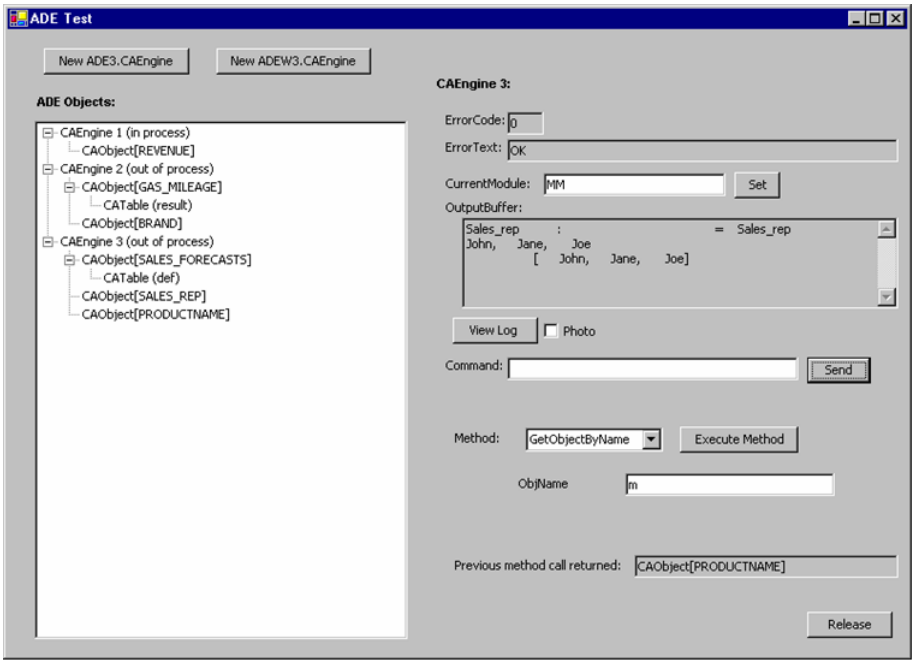
1. From the Window's Control panel, select Administrative Tools ⇨ Component Services.
2. In the DCOM Config folder, locate "Analytica Decision Engine Local Server 4.0".
3. Select Properties from the right mouse menu, and select the Security tab.
4. Set Launch and Activation Permissions to Customize, then click on **Edit…**
5. For the user `{computer_name}\ASPNET`, grant local launch and local activation permission.
6. Save these settings. A reboot of the machine may be necessary.

When these permissions are not properly configured, a "security exception" will occur on the line of your program that attempts to instantiate the **CAEngine**.

The AdeTest Program

ADE 4.0 ships with a sample program called AdeTest.exe. The executable can be found in the Examples/AdeTest/bin directory. You can use AdeTest to exercise the functionality of either the in-process (adew.dll) or the local process (`ADE.exe`) versions of ADE 4.0. Using AdeTest, you can send script commands to the engine, create ADE objects, set or call virtually any of the properties and methods of the ADE objects. If you have Visual Studio 2005 installed, you can step through the code in the Visual Studio Debugger to observe the methods being called.

The diagram shows a screenshot of the AdeTest program. The left-hand pane shows a list of ADE objects that the program is currently holding. The right side shows details of one of those objects. In the screenshot, there are three **CAEngine** instances, each with a different model open. The first **CAEngine** is an in-process (`ADEW.DLL`) instance, while the second two are out-of-process local servers (`ADE.exe`) instances. The two buttons above the left pane can be used to create additional **CAEngine** instances, while the Release button at the lower-right corner of the right-hand panel releases an instance. The right-hand panel shows information about the third **CAEngine** instance. The current values for the **CAEngine** properties **ErrorCode**, **ErrorText**, **CurrentModule**, **OutputBuffer** and **Photo** are displayed. You can execute a typescript command by typing the command into the text box area and pressing the "Send" button. Or you can execute any of the method of **CAEngine** by selecting the method in the drop-down Method box, filling in the parameters and pressing the "Execute Method" button



If you click on an object in the left-hand pane, the properties for that object will be displayed on the right-hand side and properties can be set or its methods called. Thus, you can simulate a series of steps your program might execute through the graphical interface.

When a method returns an object, for example, as with **CAEngine::GetObjectByName**, the object returned is added to the tree on the left as a child of the object that created it. After executing a method from a class other than **CAEngine**, it is a good idea to glance at the corresponding **CAEngine**'s panel to check the **ErrorCode, ErrorText**, and **OutputBuffer** properties.

The *Photo* checkbox in the Analytica window is mirrored by the **Photo** property of the **CAEngine** class. By default the **Photo** property is `False`, so typescript communications between the client and ADE are not copied to the Analytica Log Window. Setting the **Photo** property to `True` will copy all subsequent typescript communications between the client and ADE. In Visual Basic, this would be done as follows:

```
ADE.Photo=True
ADE.Photo=False
```

Turning the **Photo** property on significantly slows down communication with ADE.

Sample Application in Excel's Visual Basic

Another example program called `excel_exam` is also included in the ADE package. The program, Analytica.xls, in the `excel_exam` directory can be loaded into Microsoft Excel and executed as a macro. This program demonstrates the use of Visual Basic for Applications in Excel for ADE communications. This sample makes use of the local server version of ADE.

Sample ASP Web Application

The example in **asp_exam** demonstrates the use of ADE from an Active Server Pages web application. This application produces a hierarchical outline of your model structure in HTML. The readme.txt file in that directory contains instructions for configuring the web server to run the example.

When using Microsoft's ASP, we recommend that you use the local server. By using the local server (**`ADE.exe`**), you can ensure that each web application, or even each session, uses a different version of **`ADE.exe`**. Currently, there is a limitation in ADE that prevents creation of two or more in-process server objects at the same time. Therefore, if you expect to have more than one session of ADE active at one time (as is almost always the case in web-based applications), always use the local server of ADE.

Using the ADE COM-interface

From a .Net project in Visual Studio 2005

From a Visual Basic, C#, J#, ASP.NET, or C++/CLR project in Visual Studio 2005, you gain access to ADE by adding a reference to it into your project. The same technique holds with slight variations in details in older (pre-.NET) versions of Visual Basic and several other non-Microsoft development environments.

In Visual Studio 2005, select "Add reference…" or "References…" from the Project menu, and in the dialog that appears, select the "COM" tab (in VC++ you'll need to press the Add new reference…" button to get to the COM tab). In the list of components, locate and select one of:

```
Analytica Decision Engine Local Server 4.0
Analytica Decision Engine Server 4.0
```

For out-of-process **`ADE.exe`** server, select the Local server, to use Adew.dll select the (non-local) server. It is also possible to add both references into a project (the AdeTest example does this), although the need for this would be rare.

The ADE classes will exposed in the name space ADE or ADEW for the local server and in-process server respectively. For convenience, you can add a using declaration to the top of your source files such as:

```
Imports ADE    ' Visual basic
using ADE;    // C#
using namespace ADE;    // C++/CLR
import ADE.*;    // J#
```

Of course, when using the in-process server you would type ADEW in place of ADE above. These declarations allow you to refer to **CAEngine, CAObject**, *etc*., in your code, rather than **ADE.CAEngine, ADE.CAObject**, *etc*., which in turn makes it very easy to convert from the local to the in-process ADE server should the need arise.

To begin using ADE, you will need to obtain a first instantiation of a **CAEngine**. This is done with one of the following lines:

```
dim ADE as CAEngine = new CAEngineClass    ' VB
CAEngine ADE = new CAEngineClass();        // C#, J#
CAEngine^ pAde = gcnew CAEngineClass();   // C++/CLR
```

CAEngine is the name of a particular abstract interface, while **ADEW.CAEngineClass** and **ADE.CAEngineClass** are the names of two particular object classes that implement that interface. The **CAEngineClass** object is the only object that you can create directly – all other ADE object instances are obtained by calling methods on existing objects.

To keep the use of the COM interface, always declare your variables with the class names **CAEngine, CAObject, CATable, CAIndex** and **CARenderingStyle**. Avoid assigning object instances to variables declared as **System.Object**. This allows the compiler to perform early binding and type checking.

Releasing Objects in .NET

In pre-.NET Visual Basic and Scripting Languages, the programming environment automatically ensures that COM objects are released immediately. This is not the case in VB.NET, ASP.NET, or other .NET programs. From .NET, it is important that your program explicitly releases each COM object when it is through with it. Setting a pointer to Null (or Nothing) is not sufficient, since the actual release doesn't occur until the next garbage collection.

To release a COM object from a .NET program, you need to execute code such as (C# syntax shown):

```
System.Runtime.InteropServices.Marshal.ReleaseComObject(ADE);
ADE = null;
```

Releasing objects in this fashion is especially important when you are using an out-of-process COM server (*e.g.*, **ADE.CAEngine**). In this case, the memory resources are predominantly consumed in the ADE process, not in your program's process. This can cause the ADE process to run out of memory before your program's process uses enough memory to cause an automatic garbage collection to occur. From a .NET-based web-application, old **ADE.EXE** processes will linger long after a session has finished unless you explicitly release the **CAEngine** object.

This need to release COM objects is not unique to ADE. You must take care to release any COM object, including those provided by Microsoft, especially when those COM objects are out-of-process.

Because of this absence of deterministic destruction in .NET, it is extremely tedious to ensure that every COM object is released. Therefore, you may also want to occasionally force an explicit garbage collection in your code, which will release all unused objects. This can be accomplished by calling

```
System.GC.Collect();
```

From an ATL Project in C++

To use ADE 4.0 from a non-Dot Net C++ project, place the following two lines at the top of your source file:

```
#import "ADE.exe"
using namespace ADE;
```

or to use the in-process server, use these line:

```
#import "Adew.dll"
using namespace ADEW;
```

You will need to include the ADE Home directory on your include path in the project settings, or spell out the complete path in the #import declaration.

Next, in your code obtain the first instance to an ADE engine using:

```
CoInitialize(NULL);
CAEnginePtr pAde(__uuidof(_CAEngine));
.
.
.
CoUninitialize();
```

CoInitialize( ) is a windows system call that is required before the COM system can be used.

If your project spans multiple code files, use

```
#import "ADE.exe" no_implementation
```

in each of your source files (or once in stdafx.h), and then in one file only (e.g., stdafx.cpp), include the line

```
#import "ADE.exe" implementation_only
```

Using the ADE Automation-interface

VBScript is an example of a scripting language, usable from Windows Scripting Host (CScript.exe or WScript.exe), pre-.NET versions of Active Server Pages, Internet Explorer, etc. JScript is another, and many other scripting OLE-Automation compliant scripting languages are available including Perl, *etc*.

These scripting languages support ActiveX Automation scripting but not COM interfaces. ADE can be used from these, often with no additional tools beyond a simple text editor, using the Automation interface.

For ADE releases prior to 4.0, the automation interface was the preferred convention to use. For languages that support direct COM calls, the COM convention is now recommended in ADE 4.0. Using Automation from C++ or C# is rather tedious and not covered here.

From Visual Basic or VBScript

To use the Automation interface, it is not necessary to add a reference to your Visual Basic project. The syntax here is similar in other scripting languages. In Visual Basic, the code to instantiate a **CAEngine** is

```
dim ADE as Object
ADE = CreateObject("ADE4.CAEngine")
```

In VBScript, and some older versions of Visual Basic, the set keyword is required:

```
dim ADE
set ADE = CreateObject("ADE4.CAEngine")
```

For the in-process server, you will use send the parameter **ADEW4.CAEngine** to the **CreateObject** call.

ADE Typescript: Command Language Communication

The `Command` property and `Send` method of the **CAEngine** class allow you to use typescript commands, sent as ASCII strings to the Engine, and receive the resulting output as another ASCII string. You may want to use a typescript command instead of an API method if:

You want to perform your own parsing on ADE output (e.g., on tabular data that are output from the Analytica Decision Engine as text strings of comma-delimited text).

No appropriate API method exists.

You perform three steps to send a typescript command to ADE:

Assign a text string containing the command to the `Command` property of your **CAEngine** object.

Use the `Send` method to send the command to the Engine. If the `Send` method returns `True`, then the command was processed without error by ADE.

Store the error code and error text (if the return code is nonzero). These two pieces of information are stored in the **CAEngine** properties **ErrorCode**, and **ErrorText**.

Get the output by calling the **OutputBuffer** function in the **CAEngine** class.

These steps are demonstrated here for various programming languages. After this simple example, subsequent example will be given using a Visual Basic syntax, but the reader should have no problem extrapolating the syntax to their language of choice.

In Visual Basic

```
Imports ADE

Module Module1
Sub Main()
    Dim Result,ErrT As String
    Dim ErrCode as Integer

    dim ADE as CAEngine = new CAEngineClass
    ADE.Command = "news" 'any typescript command
    dim SendCode as Boolean = ADE.Send
    If SendCode = False Then
        ErrCode = ADE.ErrorCode
        ErrT = ADE.ErrorText
    Else
        Result = ADE.OutputBuffer
    End If
End Sub
End Module
```

In VBScript

```
set ADE = CreateObject("ADE4.CAEngine")
ADE.Command = "news"
If ADE.Send = False Then
    ErrCode = ADE.ErrorCode
```

```
    ErrT = ADE.ErrorText
Else
    Result = ADE.OutputBuffer
End if
```

In C#

```
using System;
using ADE;
namespace ADE_from_Csharp
{
    class Program
    {
        static void Main()
        {
            String errT, result;
            int errCode;
            CAEngine ADE = new CAEngineClass();
            ADE.Command = "news";
            if (!ADE.Send()) {
                errCode = ADE.ErrorCode;
                errT = ADE.ErrorText;
            } else {
                result = ADE.OutputBuffer;
            }
        }
    }
}
```

In J#

```
import ADE.*;
public class Program
{
    public static void main( )
    {
        String errT, result;
        int errCode;
        ADE.CAEngine ADE = new ADE.CAEngineClass();
        ADE.set_Command("news");
        boolean sendRes = ADE.Send();
        if (!sendRes) {
            errCode = ADE.get_ErrorCode();
            errT = ADE.get_ErrorText();
        } else {
            result = ADE.get_OutputBuffer();
        }
    }
}
```

In C++/CLR

```
using namespace System;
using namespace ADE;
void main( )
{
    String ^result, ^errT;
    int errCode;
    CAEngine^ ADE = gcnew CAEngineClass();
    ADE->Command = "news";
    if (!ADE->Send()) {
        errCode = ADE->ErrorCode;
        errT = ADE->ErrorText;
    } else {
        result = ADE->OutputBuffer;
    }
}
```

In VC++ (without .NET)

```
#import "ADE.exe"
using namespace ADE;
void main( )
{
    CoInitialize(NULL);
    _bstr_t errT, result;
    int errCode;
    _CAEnginePtr pAde(__uuidof(_CAEngine));
    pAde->Command = "news";
    if (!pAde->Send()) {
        errT = pAde->ErrorText;
        errCode = pAde->ErrorCode;
    } else {
        result = pAde->OutputBuffer;
    }
    CoUninitialize();
}
```

Errors and Error Handling

The **CAEngine** properties **ErrorCode** and **ErrorText** should be queried after any operation with ADE whenever an error is possible. Reading a value of a property from an ADE object does not change the error code. Setting the value of a property may result in an error code, usually indicating an illegal value for that property. All method calls will reset the **ErrorCode** to zero if there is no error, or to a value indicating the error.

To get additional information on an error, check the **OutputBuffer** property of **CAEngine**. Any error messages that a user of Analytica would have seen will appear in the output buffer.

Chapter 5

Working with Models, Modules, and Files

This section contains examples of various common operations and manipulations you might perform on objects in your Analytica model.

Working with Models, Modules, and Files

Models and Modules

Note: *In VBScript, VBA and pre-dot-NET versions of Visual Basic, the Set keyword was necessary when assigning an object to a variable. In VB.NET, the Set keyword is no longer necessary. The Set keyword is not used in the examples below.*

- To create a new model:

```
If ADE.CreateModel("NewModelName") Then
    'Model successfully created
End If
```

The **CreateModel** method only requires one parameter, a string containing a model name.

- To open an existing Analytica model:

```
Dim ModName as String
ModName = ADE.OpenModel("C:\ ... \Anamodel.ana")
If ModName="" then
    ' Handle Error condition here
End if
```

If a model has already been opened, that model will be closed automatically before the new model is created. If the specified filename is not legal, **OpenModel** will return an empty string. In that case, use the **ErrorCode** property of **CAEngine** to determine the cause of the error. Be aware that an `ErrorCode=2` warning is often returned even though the load is successful. For full details as to what has caused an error or warning, use the **OutputBuffer** property of the **CAEngine**. You must use the backward slash (\) for the path delimiter when using ADE. It does not support the forward slash (/).

- To add a module from a file to the currently open model:

```
Dim Merge as Boolean = True
Dim ModName as String
ModName = ADE.AddModule ("C:\...\MyLibrary.ana", Merge)
if ModName="" Then
    ' Handle error conditions here
End if
```

The **FileSpec** parameter should contain the path and filename of the module to be included. The `Merge` parameter is a Boolean variable that determines whether preexisting objects with identical names are overwritten. If `Merge=True` then conflicting variables are overwritten. If `Merge=False`, and there are conflicting variables, then the call to **AddModule** will fail.

- To read a script file:

```
If ADE.ReadScript("C:\..\MyScript.ana") Then
    ' Script successfully read
End If
```

A script file can contain a list of typescript commands. Upon loading the file, the Engine will execute the commands contained in the file. Errors encountered while running the script file are described in the `ErrorText` property.

- To save a module (i.e., a subset of the current model) in a separate file:

```
If AdeSaveModuleFile ("MyLibrary", "C:\...\MyLibrary.ana") Then
    ' Save succeeded
End If
The first parameter is the module identifier, the second is the file
name.
```

- To save the current model in a file:

```
If ADE.SaveModel("C:\...\MyNewModel.ana") Then
    ' Save succeeded
End If
```

- To close the current model without saving:

```
If ADE.CloseModel() Then … ' Close succeeded
```

The CloseModel method takes no parameters.

ADE Objects

- To create a new **CAObject** object:

```
Dim ObjName As String = "NewVariable"
Dim ObjClass As String = "Variable"
Dim var As CAObject = ADE.CreateObject(ObjName, ObjClass)
```

The object name and the class of the object to be created are passed into the `CreateObject` method. Note that an identifier and not the title of the object should be used when giving the object a name. Most Object-related methods use their **Identifier** attribute, not their **Title** attribute. ADE can create the following types of objects: Variable, Module, Chance, Constant, Decision, Index, and Objective. Refer to the *Analytica User Guide* for more information on these object types.

- To delete an Analytica object from a model:

```
Dim obj as CAObjec
If ADE.DeleteObject(obj) Then … ' Successful
```

- To set the active module:

```
ObjName = "ModuleToMakeActive"
ObjClass  =  "Module"
Var = ADE.CreateObject (ObjName, ObjClass)
ADE.CurrentModule = Var
```

ADE utilizes a hierarchy to order objects. When an object is created, it is created inside the current module. By default, all objects are placed within the top-level module unless you set the **CurrentModule** property.

- To identify the current module:

```
Dim module As CAObject = ADE.CurrentModule
```

- To obtain a **CAObject** object when you know the name of an Analytica variable (this is probably the most commonly used method in ADE):

```
Dim Var As CAObject = ADE.GetObjectByName ("IdentifierInModel")
If Var Is Nothing Then
    'Analytica model associated with Ana
    'does not contain variable with
    'identifier "IdentifierInModel"
End If
```

The method **CAObject::Get** is synonymous with **GetObjectByName**.

- You can get all Analytica object attributes, except the **Identifier,** using the **GetAttribute** method:

  ```
  UnitsOfVar = Var.GetAttribute ("Units")
  ```

  Use the `SetAttribute` method to change an Attribute of an Analytica object (except for **Identifier**):

  ```
  If Var.SetAttribute ("definition","A/B") Then
      'Attribute Set Correctly
  Else
      'Attribute Not Set
  End If
  ```

- To access or rename the Identifier of an object, use the **Name** property:

  ```
  Dim oldName As String = Var.Name
  Var.Name = "NewIdentifer"
  ```

For the full lists of object attributes see *The Analytica Scripting Guide* chapter 3, "Objects and their attributes."

Retrieving Computed Results

The **CAObject** class contains three methods that cause results to be computed and returned. The **Result** method evaluates an object in your model and returns the result as a single value. This is most useful if you know that the result will be a single number or single text string. The **ResultTable** method evaluates an object in your model and returns the result as a **CATable** object. Methods and properties of the **CATable** object allow you to understand what dimensions are present and to access individual elements (cells). The **Evaluate** method processes an arbitrary expression and returns the result of parsing and evaluating that expression as a multi-dimensional **CATable**.

When retrieving results you have control over which computation mode is used to compute the result. You can compute the deterministic mid point value, or the various probabilistic views: Mean, Sample, PDF, CDF, Statistics or Bands. Set the **ResultType** to indicate which result type you desire (default is Mid).

Whether you are computing a scalar or a table, your program will eventually access individual “atomic” values such as numbers or text strings. You can use various **RenderingStyle** settings to control the form in which these values are returned. For example, numeric values can be returned as floating point numbers, formatted strings, or full-precision string depictions. Textual strings can be returned with or without surrounding quotes.

- To evaluate and obtain a simple result (e.g., a scalar) of an object use the **Result** method of **CAObject**:

```
Dim Obj As CAObject
Dim Result
Obj = ADE.GetObjectByName ("ObjectToEvaluate")
Result = Obj.Result
If ADE.ErrorCode = 0 Then
    'Result was successfully retrieved
Else
    'An error occurred
End If
```

The **Result** property of **CAObject** retrieves, by default, the midpoint result of the object. It will return the result as a variant (or in .NET, as a System.Object). This method is convenient for retrieving the results of objects that evaluate to a scalar.

- To evaluate and obtain the result of an object as something other than the midpoint use the **ResultType** property of **CATable**:

```
Dim Obj As CAObject = ADE.GetObjectByName("ObjectToEvaluate")
Dim Result
Obj = ADE.GetObjectByName("ObjectToEvaluate")
Obj.ResultType = 1  ' get result as mean
Result = Obj.Result
If ADE.ErrorCode = 0 Then
    'Result was successfully retrieved as a mean
Else
    'An error occurred
End If
```

The **ResultType** property is used to indicate the type of result that **Result** should return. Possible values are: (0=Mid point, 1=Mean, 2=Probabilistic Sample, 3=PDF, 4=CDF, 5=Statistics, 6=Probability Bands). When ResultType>=2, the result will always be a table, even if the mid and mean are scalars. See the next section for a discussion on retrieving table results.

- To retrieve a formatted result, set properties of the object's RenderingStyle.

```
Dim Obj As CAObject = ADE.GetObjectByName("ObjectToEvaluate")
Dim Result
Obj.RenderingStyle.NumberAsText = true
Obj.RenderingStyle.StringQuotes = 2  ' double quotes.
Result = Obj.Result
If ADE.ErrorCode = 0 then
    ' Result was successfully returned.
End If
```

In this example, numbers will be returned as formatted text using the object's number format property. Strings will be returned surrounded by double quotes. So, for example, the numeric value 1.2K might be returned as the string "$1,200.00" if the number format happens to be fixed point, 2 digits, with trailing zeros, thousand separators and currency. This numeric value is returned as a text string because the NumberAsText property is True. The string would be returned as "`"$1,200.00"`" with two extra double quote characters in the result string. This is controlled by the StringQuotes property (0 = no quotes, 1 = 'single quotes', 2=`"`double quotes`"`).

Retrieving Multi-Dimensional Results

Before delving into the details of how to obtain results from table objects (arrays with one or more dimensions), let us briefly discuss the conceptual model of a table in Analytica.

An Analytica table has the following components:

- Indexes, each of which identifies a dimension of the table
- Values in the cells of the table
- Index labels, which identify the coordinates of each cell

The number of indexes determines the dimensionality of the table. So, for example, if a table contains two indexes, then the table is 2-dimensional.

The number of elements in the index determines the actual number of cells in the table. Suppose table **T** is composed of 2 indexes, **I** and **J**. If **I** has 5 elements (**AA, BB, CC, DD, EE**) and J has 3 elements (**A, B, C**), then **T** is either a 5x3 table, or a 3x5 table, depending on your perspective.

Determining your perspective of a table is very important when working with ADE. It is up to you to tell ADE how you wish to view the table. So, for example, in the above paragraph, if you tell ADE to use index **I** first, followed by index **J**, then element 2,3 would be the element described by position **I=B, J=CC**. If, however, you tell ADE to use index **J** first, followed by index **I**, then element 2,3 would be described by position **I=C, J=BB** (note that tables in ADE are 1-based; that is, each dimension goes from 1 to *N* where *N* is the size of the index). The method called **SetIndexOrder**, described below, allows you to set the order of the indexes for your table, so that you can look at the table in any way you desire.

The ADE methods are very flexible in terms of how you refer to individual elements in the table. You can either refer to the individual elements by their position number or by their label names. So, for example, you can tell ADE to give you the element at position 2,1 (2 along the first index, and 1 along the second index), or you can tell ADE to give you the element described by '**BB**','**A**' where '**BB**' and '**A**' are label names in their respective indexes. The methods most commonly used for these types of transactions (**GetDataByElements** and **GetDataByLabels**) are described below.

As discussed in the previous section, the **Result** and **ResultType** methods are used to evaluate and obtain the result of an object. For objects that evaluate to multi-dimensional results, however, it is often inconvenient to use the **Result** method. After all, the output of the **Result** method for a multi-dimensional result would be a long comma delimited string in the following form:

```
Table Index1...IndexN [Value1, Value2...]
```

Here, **Index1** to **IndexN** are the indexes of the table, and Value1 to ValueN are the values in the table (which are filled in row by row). So, if we wanted to get at a particular element in the table after using the **Result** method, we would have to parse through the comma delimited string, returned from **Result**, in order to get at the element of interest. Fortunately, ADE provides an ADE object of type **CATable** that provides methods to simplify the manipulation of tables.

- To evaluate and obtain the result of an object as a table use the **ResultType** method of **CAObject**:

```
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim TableResult As CATable = Obj.ResultTable
```

```
If Not (TableResult Is Nothing) Then
'Result table was successfully retrieved
Else
'An error occurred, or result is scalar
End If
```

The **ResultTable** method of **CAObject** returns an automation object of type **CATable**. **CATable** contains various methods that allow you to set, retrieve, and manipulate individual elements in the table. The first thing that you will, more than likely, want to do after retrieving the **CATable** object, is to set the index order of the result table.

- To parse and evaluate an arbitrary expression, use the **Evaluate** method of **CAObject**.

```
Dim Obj As CAObject = ADE.GetObjectByName ("ContextObject")
Obj.ResultType = 2    ' Sample
Dim TableResult As CATable = Obj.Evaluate("Normal(X,Y^2) / Z")
If ADE.ErrorCode <> 0 Then
    'An error occurred
Else
    'Evaluation successful
End If
```

To use **Evaluate**, you must first obtain a **CAObject** instance. Although the expression you are evaluating may have nothing to with any specific object, the **CAObject** serves a couple of purposes. First, the **ResultType** property of the object provides a place to specify the result type that you want computed. Second, if you make use of the **NumberAsText** rendering style, the number format stored with the indicated object will determine how the numbers are formatted. Often, however, the object you use is of no consequence – you can even use the top-level model object as your context object.

Comparing the previous two examples demonstrates also that there are often two ways to detect failure. The **ErrorCode** property is non-zero if an error occurred during the evaluation of a method. And for many methods, the return value is `Nothing` or `False` if it fails. .

- To set the index order of a **CATable** object, use the **SetIndexOrder** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) as String
Set Obj = ADE.GetObjectByName ("MultiDimObject")
Set TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"

    If TableResult.SetIndexOrder(IndexOrder)Then
        'Index Order set successfully
    Else
        'An error occurred in setting index order
    End If
Else
 'An error occurred, or result is scalar
End If
```

The above code assumes that we are manipulating a two-dimensional table. We set the index order of this table so that **Index2** is the first index, and **Index1** is the second index.

In some computer languages, the first element of an array is considered position 0 (zero-based), and others it is position 1 (one-based). Analytica's Slice function, and the ADE methods are one-based . Older versions of Visual Basic are one-based, while current versions of Visual Basic and most other modern programming languages are zero-based. In the above example, the Visual Basic array was declared and used as follows

```
Dim IndexOrder(2) As String
IndexOrder (1) = "Index2"
IndexOrder (2) = "Index1"
```

In the modern VisualBasic, this declares an array that ranges from position 0 to position 2 – an array having three elements. Because the first element was not set, it will contain the special value Empty. ADE can recognize whether zero-based or one-based arrays are being passed to it. So, depending on your preference, it would work equally well to use a zero-based version, *i.e.*

```
Dim IndexOrder(1) As String
IndexOrder (0) = "Index2"
IndexOrder (1) = "Index1"
ResultTable.SetIndexOrder(IndexOrder)
```

- To retrieve an element in a table by index order use the **GetDataByElements** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) As String
Dim Pos (2) As Integer
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    RetValue = TableResult.SetIndexOrder (IndexOrder)
    If RetValue = True Then
        'Index Order set successfully
        Pos (1) = 2
        Pos (2) = 1
        Element = TableResult.GetDataByElements (Pos)
        If ADE.ErrorCode = 0 Then
            'element retrieved successfully
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
```

```
    'An error occurred, or result is scalar
End If
```

The above code uses **GetDataByElements** to retrieve the element at position **Index2**=2, **Index1**=1 and stores the result to **Element**.

- To retrieve an element in a table by index labels use the **GetDataByLabels** method:

```
Dim Obj As CAObject
Dim TableResult As CAObject
Dim IndexOrder (2) As String
Dim Pos (2) As String
Dim Element
Set Obj = ADE.GetObjectByName ("MultiDimObject")
Set TableResult = Obj.ResultTable
If Not TableResult Is Nothing Then
    'Result table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TableResult.SetIndexOrder(IndexOrder) Then
        'Index Order set successfully
        Pos (1) = "SomeLabelInIndex2"
        Pos (2) = "SomeLabelInIndex1"
        Element = TableResult.GetDataByLabels (Pos)
        If ADE.ErrorCode = 0 Then
            'element retrieved successfully
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred, or result is scalar
End If
```

The above code uses **GetDataByLabels** to retrieve the element at position **Index2**="SomeLabelInIndex2", **Index1**="SomeLabelInIndex1" and stores the result to **Element**.

- To control the format of elements obtained by **GetDataByLabels**, **GetDataByElements**, AtomicValue or **GetSafeArray** methods, set the **CATable**'s RenderingStyle properties:

```
Set TableResult = Obj.ResultTable
Set rs As CARenderingStyle = TableResult.RenderingStyle
rs.NumberAsText = True
rs.FullPrecision = True
rs.UndefValue = ""
rs.StringQuotes = 1
Dim Element
If TableResult.SetIndexOrder(Split("Index2;Index1",";")) Then
    Element = TableResult.GetDataByLabels( _
    Split("SomeLabel1,SomeLabel2",","))
    If Add.ErrorCode=0 Then
        ' Element retrieved successfully
```

```
    End If
End If
```

- To retrieve the whole table into a Visual Basic or .NET array in one call use the **GetSafeArray** method:

```
Dim Obj As CAObject
Dim TableResult As CATable
Dim IndexOrder (2) as String
Dim Pos (2) As Integer
Dim TheWholeTable
Obj = ADE.GetObjectByName ("MultiDimObject")
TableResult = Obj.ResultTable
If Not (TableResult Is Nothing) Then
    'Result table was successfully retrieved
IndexOrder (1) = "Index2"
IndexOrder (2) = "Index1"
If TableResult.SetIndexOrder(IndexOrder) Then
    'Index Order set successfully
TheWholeTable = TableResult.GetSafeArray
If ADE.ErrorCode = 0 Then
    'table retrieved successfully
Else
    'an error occurred
End If
Else
    'An error occurred in setting index order
End If
Else
    'An error occurred, or result is scalar
End If
```

The above code uses **GetSafeArray** to store the entire table in **TheWholeTable**. The elements of each dimension associated with the table returned from **GetSafeArray** are indexed 1 to *N*, where *N* is the length of the dimension. The lower bound of the Safe Array can be changed to zero using:

```
TheWholeTable.RenderingStyle.SafeArrayLowerBound = 0
```

prior to calling **GetSafeArray**. The syntax for reading a multi-dimensional result in a .NET array in C# is worth mentioning:

```
Array theWholeTable = (Array) tableResult.GetSafeArray( );
```

- To determine the number of dimensions of the table, use the **NumDims** property:

```
NumDimensions = ADE.Get("MultiDimObject").TableResult.NumDims
```

- To get the index names associated with the table, use the **IndexNames** method:

```
Dim CurIndexName As String
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim TableResult As CATable = Obj.ResultTable
Dim NumDimensions As integer = TableResult.NumDims
Dim I as Integer
For I = 1 To NumDimensions
CurIndexName = TableResult.IndexNames(I)
```

```
MsgBox "Current index is " & CurIndexName
Next I
```

The **IndexNames** method returns the index names of the table in the order specified to **SetIndexOrder**. If **SetIndexOrder** has not been set for the **CATable**, then the default order of the indexes will be returned.

- To get the **CAIndex** objects associated with your table, use the **GetIndexObject** method of **CATable**:

```
Dim CurIndexName As String
Dim IndexObj As CAindex
Dim Obj As CATable = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CATable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames(NumDimensions)
Dim IndexObj As CAObject = Res.GetIndexObject(CurIndexName)
```

The above example retrieved the last **CAIndex** object, with respect to the index order, from the table. The **CAIndex** object provides properties and methods that allow you to obtain information about the respective index.

- To get the number of elements in the index, use the `IndexElements` property:

```
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CATable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames (NumDimensions)
Dim IndexObj As CAIndex = Res.GetIndexObject(CurIndexName)
Dim NumElsInIndex As Integer = IndexObj.IndexElements
```

- To get an index label at the specified position in the index, use the **GetValueByNumber** method:

```
Dim I As Integer
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CATable = Obj.ResultTable
Dim NumDimensions As Integer = Res.NumDims
Dim CurIndexName As String = Res.IndexNames(NumDimensions)
Dim IndexObj As CAIndex = Res.GetIndexObject (CurIndexName)
Dim Str As String = "The elements in the index are: " & vbCrLf
For I=1 To IndexObj.IndexElements
    Str = Str & IndexObj.GetValueByNumber(I) & " "
Next I
MsgBox Str
```

- To get at the position of an index label in an index, use the `GetNumberByValue` method:

```
Dim Obj As Object
Dim NumDimensions
Dim TableResult As Object
Dim CurIndexName As String
Dim IndexObj As Object
Dim I As Integer
Dim IndexPosition As Integer
Dim Obj As CAObject = ADE.GetObjectByName ("MultiDimObject")
Dim Res As CATable = Obj.ResultTable
Dim IndexName As String = Res.IndexNames(TableResult.NumDims)
```

```
Dim IndexObj As CAIndex = Res.GetIndexObject (IndexName)
Dim IndexPosition As Integer
IndexPosition = IndexObj.GetNumberByValue("SomeIndexLabel")
If ADE.ErrorCode = 0 Then
    'the index position was successfully retrieved
Else
    'an error occurred
End If
```

- Obtain the scalar value in a zero-dimensional array.

  Sometimes, it is not possible to know in advance whether the evaluation of an object will return a multi-dimensional result or a scalar. In this case, use **ResultTable**. If the result happens to be a scalar, **NumDims** will return zero. In this case, the so-called "array" isn't an array at all, but rather contains a single atomic value. It is also possible to end up with a zero-dimensional array after calling the **CATable::Slice** or **CATable::Subscript** methods. To obtain the atomic value, use the **CATable::AtomicValue** method.

```
Dim Res As CATable = ADE.Get("SomeObject").ResultTable
Dim x As Object
If Res.NumDims = 0 Then
    x = Res.AtomicValue
Else
    ' Handle the array case.
End If
```

- Dimensionality Reducing Slice and Subscript Operations:

  The **Slice** and **Subscript** methods of **CATable** return a new **CATable** object with the number of dimensions reduced by one. These methods are similar to the Slice and Subscript functions built into Analytica. Slice returns the Nth slice (by position) along a given dimension. **Subscript** returns the slice corresponding to a specified index value.

```
Dim PandL CATable = ADE.Get("P_n_L_Statement").ResultTable
Dim CatIndex As CAIndex = Res.GetIndexObject("Categories")
Dim Expenses As CATable = Res.Subscript(CatIndex,"Expenses")
Dim Year As CAIndex = Expenses.GetIndexObject("Year")
Dim InitialExpense As CATable
InitialExpense = Expenses.Slice(Year,1).AtomicValue
```

Creating Tables and Setting Values in Tables

We can apply to definition tables the same methods described above to retrieve values from result tables. A ***definition table***, as the name suggests, is when the definition of an object is a **Table** function (also known as an **Edit table** in Analytica). Please note:

The value of an Analytica variable (accessed via ResultTable) may be an array not because it was defined by a Definition Table but simply because it is defined as an expression or function that returns an array value.

When using an edit table, you need to pay careful attention to whether you are passing general expressions into each table cell, or just literal strings. The RenderingStyle.GeneralExpression property determines how string values that you

send to the table are interpreted. By default, GeneralExpression=true, which means that if you set a cell value to the string "Revenue", this is an actual expression consisting of one variable identifier, and not a literal string. If you are populating a definition table with literal constants (as you might an input table to your model), you should either use RenderingStyle.GeneralExpressions=false, or remember to prepend and append quotation marks on all literal string values.

An object defined as a definition table will not necessarily produce the same table when **ResultTable** is called. After all, the definition table can be defined to be an array of identifiers. When **ResultTable** is called, each identifier's result will be evaluated, and a new table will be produced which would be different than the definition table. If identifiers evaluate to arrays, the result table may have more dimensions than the definition table.

- To get the definition table of an object as a **CATable**, use the **DefTable** method of **CAObject:**

```
Dim Obj As Object
Dim TableDef As Object
Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
If Not TableDef Is Nothing Then
'Definition table was successfully retrieved
Else
'An error occurred, or definition is not a table
End If
```

  Once the definition table is retrieved, we can use all the same methods described in the above section (**GetDataByElements**, **GetDataByLabels**, `SetIndexOrder`, *etc*.) to retrieve elements in the table and to obtain information about the indexes in the table. We can also use the same method that we used above in determining whether the result of the object was multi-dimensional or scalar to determine whether the definition of the object is a table or scalar:

```
Dim Obj As Object
Dim TableDefinition As Object
Dim ScalarDefinition

Obj = ADE.GetObjectByName ("SomeObject")
TableDefinition = Obj.DefTable
If TableDefinition Is Nothing Then
    ScalarDefinition = Obj.GetAttribute ("definition")
    If ADE.ErrorCode = 0 Then
        'you have a scalar definition
    Else
        'an error occurred
    End If
Else
    'you have a table definition
End If
```

- To set an element in a table by index order use the **SetDataByElements** method of **CATable**:

```
Dim Obj As CAObject
Dim TableDef As CATable
Dim IndexOrder (2) As String
```

```
Dim Pos (2) As Integer
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
If Not TableDef Is Nothing Then
    'Definition table was successfully retrieved
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TableDef.SetIndexOrder (IndexOrder) Then
        'Index Order set successfully
        Pos (1) = 2
        Pos (2) = 1
        Element = "'ABC'"    ' Notice the extra quotes

        If TableDef.SetDataByElements(Element,Pos) Then
            'element successfully set
            If TableDef.Update Then
                'model successfully updated
            Else
                'error updating def in model
            End If
        Else
            'an error occurred
        End If
    Else
    'An error occurred in setting index order
    End If
Else
    'An error occurred, or definition is scalar
End If
```

The above code uses **SetDataByElements** to set the element at position **Index2**=2, **Index1**=1 to **Element**. Note the use of the quotes around **ABC**. Here, since **ABC** is single quoted, we are putting the string "ABC" in the table. If we instead set **Element** to "ABC", then the expression `ABC` would be placed in the table. In the latter case, **ABC** would, more than likely, be a variable. If an identifier, **ABC**, did not exist in the model, then an error would have occurred while trying to set the element in the latter case. The code then used **Update** to update the model with the new definition. It is important to note that the model containing the object will not be updated until **Update** is called. Therefore, if **Update** is not called, and the result of a node that depends on this object is later calculated, the old definition of this object will still be used. The other important thing to note is that **Update** functions very differently for result tables as it does for definition tables. For result tables, **Update** will retrieve the result from the specified object again. It will, therefore, overwrite any changes that were made to the object using **SetDataByElements** and **SetDataByLabels**.

- To set an element in a table by index labels use the `SetDataByLabels` method of **CATable**:

```
Dim Obj As CAObject
Dim TabDef As CATable
Dim IndexOrder (2) as String
```

```
Dim Pos (2) as String
Dim Element
Obj = ADE.GetObjectByName ("MultiDimObject")
TabDef = Obj.DefTable
If Not TabDef Is Nothing Then
    'Definition table was successfully retrieved
    TabDef.RenderingStyle.GeneralExpression = False
    IndexOrder (1) = "Index2"
    IndexOrder (2) = "Index1"
    If TabDef.SetIndexOrder (IndexOrder) Then
        'Index Order set successfully
        Pos (1) = "SomeLabelInIndex2"
        Pos (2) = "SomeLabelInIndex1"
        Element = "ABC"
        If TabDef.SetDataByLabels(Element,Pos) Then
            'element set successfully
            If TabDef.Update Then
                'model successfully updated
            Else
                'an error occurred
            End If
        Else
            'an error occurred
        End If
    Else
        'An error occurred in setting index order
    End If
Else
    'An error occurred, or definition is scalar
End If
```

The above code uses `SetDataByLabels` to set the element at position **Index2**="SomeLabelInIndex2", **Index1**="SomeLabelInIndex1" to **Element**. In this example, the **RenderingStyle.GeneralExpression** property was set to `False`. This eliminates the need to explicitly include quotes around the string as was done in the previous example for **SetDataByElements**.

- To set the whole table in one call, use `PutSafeArray` and `Update`:

```
Dim Obj As Object
Dim TableResult, TableDef As Object
Dim RetValue
Dim TheWholeTable

Obj = ADE.GetObjectByName ("MultiDimObject")
TableDef = Obj.DefTable
TableResult = Obj.ResultTable
TheWholeTable = TableResult.GetSafeArray
'make changes to TheWholeTable
…
RetValue = TableDef.PutSafeArray (TheWholeTable)
If RetValue = True Then
    'table successfully put
    RetValue = TableDef.Update
```

```
        If RetValue = True Then
            'model successfully updated
        End If
    End If
```

- To create a whole table from scratch, use `CreateDefTable`:

```
Dim Obj As Object
Dim RetValue
Dim IndexLabs (1 To 2) As Variant

Obj = ADE.CreateObject ("MyNewTable", "Variable")
IndexLabs (1) = "I"
IndexLabs (2) = "J"
RetValue = Obj.CreateDefTable (IndexLabs)
If RetValue = True Then
'a table indexed by I and J has successfully
'been created.  We are assuming that I and J
'already exist
Else
'an error occurred when creating the table
End If
```

  The above code created a definition table indexed by **I** and **J**. The table is dimensioned according to the size of **I** and **J**. All the cells in the table are initially set to 0. The user can then call **DefTable**, and then use **SetIndexOrder, SetDataByElements, SetDataByLabels, PutSafeArray**, and **Update** to put values into the table. Note that the function **CreateDefTable** is very rarely used in an ADE program. After all, it is much easier to create an object in Analytica than it is in ADE.

Adjusting How Values Are Returned

Analytica models may contain several different data types for values in attributes or in the cells of a table or index. Data types include floating point numbers, textual strings, the special values `Undefined` and `Null`, references, and `varTerm` (handles to other objects). When these data types are returned and ultimately mapped to data types in the programming language you are using, you may want or need to alter how the values are returned. The **RenderingStyle** provides the control.

The **CAObject, CATable** and **CAIndex** objects all contain a property called **RenderingStyle**, which returns a **CARenderingStyle** object. Properties of the rendering style can be changed to change how values are returned. You can also control whether Safe Arrays returned by Analytica are 1-based or 0-based. These settings impact for **CAObject: GetAttribute** and **Result**; for **CATable: GetDataByElements, GetDataByLabels, AtomicValue,** and **GetSafeArray**; and for **CAIndex::GetValueByNumber.**

When transferring values to cells in a **DefTable**, you can also control whether the cells are populated by literal strings and values, or by general expression. This is controlled by the GeneralExpression property of **CARenderingStyle**.

The **DefaultRenderingStyle** and **DefaultDefTableRenderingStyle** properties of **CAEngine** can be set once just after the **CAEngine** has been instantiated to set the rendering style globally. For example, if you always use zero-based arrays, this can be specified once.

- Retrieving numeric values as numbers:

```
obj.RenderingStyle.NumberAsText = False
Dim x As Double = obj.Result
```

  Numeric values are returned as numbers by default, so unless NumberAsText is set to True at some point, there is no need to specify this explicitly.

- Retrieving numeric values as formatted strings:

```
obj.RenderingStyle.NumberAsText = True
obj.RenderingStyle.FullPrecision = False
Dim s As String = obj.Evaluate(1/3 * 10^6)
```

  The number format associated with obj is used to format the numeric value. A suffix style with 4 digits returns “333.3K” for this example.

- Retrieving numeric values as strings with no loss of precision:

```
obj.RenderingStyle.NumberAsText = True
obj.RenderingStyle.FullPrecision = True
Dim s As String = obj.Evaluate(1/3 * 10^6)
```

  Analytica continues to use the number format associated with obj, but the significant digits is increased so as to avoid any loss in precision. So, suffix, exponential, fixed point, and percent formats will not be truncated. If a date, integer or Boolean format is used, some truncation may still occur. In the example, the return value would be “333.333333333333”.

- Retrieving string results without quotation marks:

```
tab.RenderingStyle.StringQuotes = 0
```

- Retrieving string results with explicit quotation marks:

```
tab.RenderingStyle.StringQuotes = 1  ' for single quotes
tab.RenderingStyle.StringQuotes = 2  ' for double quotes
```

- Using a custom value for Undefined:

```
ADE.DefaultRenderingStyle.UndefValue = ""
```

  By default, the special value Undefined is returned as a the special Windows variant type Empty. There are some scripting languages that cannot deal with the Empty data type, so if you encounter this problem, you may want to change this value.

- Setting the cells of a definition table to string values:

```
defTab.RenderingStyle.GeneralExpression = False
defTab.SetDataByElements("A & B",inds)
```

  In this example, the indicated table cell is set to the string value "A & B". When this table is evaluated, this cell’s result will be a string containing the five characters "A & B".

- Setting the cells of a definition table to expressions:

```
defTab.RenderingStyle.GeneralExpression = True
defTab.SetDataByElements("A & B",inds)
```

  Here the cell is set to the expression “A & B”. When this table is evaluated, the variable named A and the variable named B will be evaluated, and their results will

be coerced to strings and concatenated by the "**&**" operator.

You can set table cells to literal strings with `GeneralExpression=True`, but you must embed explicit quotations marks in the expression. For example:

```
defTab.RenderingStyle.GeneralExpression = True
defTab.SetDataByElements("'A & B'",inds)
GeneralExpression=True by default.
```

Using the Analytica Graphing Engine

When you have a **CATable** result with at least one dimension, you can obtain a graph of the result as an image. One use of this is to embed graphs as JPEG images in a web page that uses ADE on the back end.

Obtaining the graph of a result requires the following steps:

1. Select the appropriate graph settings, such as chart type, axis range settings, colors, fonts, and so on. The easiest way is to open the model in Analytica Enterprise, and select the settings you want for each variable using the **Graph Setup…** dialog.

   The graph template you create from using the **Graph Setup…** dialog is stored in the GraphSetup attribute of the object. You can copy the GraphSetup attribute from an existing variable if you need to change the style template.
2. From ADE, obtain a **CATable** with the result to be graphed.
3. Set the **GraphWidth** and **GraphHeight** properties of the **CATable** object to indicate the desired size of the graph in pixels.
4. If your result has more than 2 dimensions, call **Slice** or **Subscript** to reduce the dimensionality to the desired dimensionality for the plot (usually one dimension without a key or two dimensions if there is a key).
5. If you have more than one dimension, call **SetIndexOrder** to select the desired pivot for the graph.
6. If sending the graph to an output stream, obtain an Windows **IStream** interface to the stream. If you have a .NET Stream (System.io.Stream), you will need to use a wrapper class (see below).
7. Call either the **GraphToStream** or **GraphToFile** methods of **CATable**, depending on where you want the graph written to. The graph can be created in different mime types (*e.g.*, "image/JPEG").

If you are able to view the result graph in Analytica with no slicers, then steps 4 and 5 are unnecessary.

- Writing a result graph to a file:

```
Dim res As CATable = obj.ResultTable
res.GraphWidth = 640
res.GraphHeight = 400
If res.GraphToFile("C:\Temp\Result.JPG","image/JPEG") Then
    ' success
End If
```

- Dynamically generating a result graph from an ASP.NET web page:

```
<%
```

```
Response.ContentType = "image/JPEG"
Dim varName As String = Request.QueryString("var")
Dim ADE As CAEngine = Session("ADE")  ' assume existing session
Dim res As CATable = ADE.Get(varName).ResultTable
res.GraphWidth = 640
res.GraphHeight = 400
Dim stream As StreamConnector = _
            new StreamConnector( Response.OutputStream)
If res.GraphToStream( stream, "image/JPEG") Then
' success
End If
%>
```

In this example code, Response, Request and Session are Active Server Page objects. The HTTP in the client browser would contain a tag such as:

```
<img src="http://www.acme.com/Myapp/graph.aspx?var=Profit">
```

When Microsoft introduced .NET, they did not make the base Stream class interface in .NET compatible with the IStream interface in Windows. Because of this, it is necessary to create a stream wrapper that implements the **IStream** interface around the .NET Stream before passing it to GraphToStream. This wrapper, class StreamConnector, is included in the example AdeTest. To use the above example, add the file StreamConnector.vb to your project.

Chapter 6 *ADE Server Class Reference*

This chapter lists the properties and methods for the five ADE server classes:**CAEngine**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle**.

ADE Server Class Reference

There are five ADE server classes: **CAEngine**, **CAObject**, **CATable**, **CAIndex**, and **CARenderingStyle.** They are listed below in that order, with a complete description of the properties and methods of each class.

Class CAEngine

Properties

Command

Description: Sets a typescript language command for execution by the ADE Automation Server. The **Send** method causes the Command to be sent to ADE for execution. For the list of typescript commands see the *Analytica Scripting Guide*.

Data type: string

Access: read/write

Usage:
```
ADE.Command = "value obj1"
```

CurrentModule

Description: The currently open module.

Data type: **CAObject**

Access: read/write

Usage:
```
Dim Obj As Object
Set Obj = ADE.CurrentModule
```

Remarks: Newly created objects are placed into the CurrentModule; so, you should set the CurrentModule before creating any new objects. Setting `CurrentModule = Nothing` means that no module is open, so all new objects will be created in the top-level Module or Model of the currently opened model.

API Errors: 44- " Module could not be set"

DefaultDefTableRenderingStyle

Description: The default rendering style controlling how definition table values are transferred to and from ADE. All definition tables returned from **CAObject::DefTable** will inherit these settings when they are first created.

Data type: **CARenderingStyle**

Access: read/write

Usage:
```
ADE.DefaultRenderingStyle.GeneralExpression = false
```

DefaultRenderingStyle

Description: The default rendering style controlling how result values are returned from ADE. All **CAObject** instances will inherit this rendering style when they are created.

Data type: **CARenderingStyle**

Access: read/write

Usage:
```
ADE.DefaultRenderingStyle.StringQuotes = 2
```

ErrorCode

Description: Returns the error code generated by the last communication with the Analytica Decision Engine Server. The property ErrorCode should be checked after setting and retrieving critical **CAEngine** properties and calling **CAEngine** methods. An ErrorCode of zero indicates the last action was successful.

Data type: integer

Access: read

Usage:
```
Dim x As Integer
x = ADE.ErrorCode
```

ErrorText

Description: short text explanation of error from ErrorCode

Data type: string

Usage:
```
Dim x As String
x = ADE.ErrorText
```

Access: read

Log

Description: A record of all Commands sent to the ADE typescript and the results received from those commands, when the **Photo** property is true

Data type: string

Usage:
```
Dim x As String
x = ADE.Log
```

Access: read

OutputBuffer

Description: A text string buffer that contains the result of the last typescript (i.e., using the Command property and Send method) interaction with the ADE.

Data type: string

Usage:
```
Dim x As String
x = ADE.OutputBuffer
```

Access: read

Photo

Description: When **Photo** is True, ADE records all typescript commands and results into the **Log** property.

Data type: boolean

Access: read/write

Usage:
```
ADE.Photo = True
```

Remarks: Setting **Photo** property to True slows down computation speed of the Engine.

Methods

AddModule(*fileName*, *merge*)

Description: Adds a module from file ***fileName*** into the **CurrentModule**. ***merge*** currently has no effect and should be set to True.

Parameters: ***fileName*** - string

merge – boolean.

Return Value: ModuleName - string

Usage:
```
ModName = ADE.AddModule ("C:\MYMOD\MYMOD.ANA", True)
```

API Errors: 39- "Module could not be found"

CloseModel

Description: Closes the model

Usage:
```
ADE.CloseModel
```

CreateObject(*objName*, *objClass*)

Description: Creates a new Analytica object with identifier ***objName*** and Class ***objClass*** in the **CurrentModule** and returns it as a **CAObject**

Parameters: ***objName*** - string

objClass – string

Return Value: **CAObject**

Usage:
```
Dim obj As CAObject
Set obj = ADE.CreateObject ("NewVar","Chance")
```

Remarks: ObjClass can be one of the following values:

Decision, Variable, Chance, Constant, Index, Module, Objective, Determ, Alias, or Formnode.

API Errors: 40- "Object could not be created"
41- "Invalid name for object"
42 - "Object name already in use"
48 – “Invalid object class”

CreateModel(*modelName*)

Description: Creates a new Analytica model with identifier ***modelName***

Parameters: ***modelName*** - string

Return Value: boolean (success or failure)

Usage:
```
boolval = ADE.CreateModel ("MyNewModel")
```

API Errors: 45- "Model could not be created"

DeleteObject(*obj*)

Description: Deletes **CAObject *obj*** from the Current Model

Parameters: **obj – CAObject**

Usage:
```
Dim Obj As CAObject
Set Obj = ADE.GetObjectByName("ObjToDelete")
ADE.DeleteObject (Obj)
```

API Errors: 41- "Invalid object"

GetObjectByName(*objName*)
Get

Description: Returns an object of type **CAObject** for an existing Analytica object with identifier ***objName***.

Parameters: ***objName*** - string

Return Value: **CAObject**

Usage:
```
Dim Obj As CAObject
Set Obj = ADE.GetObjectByName ("MyObject")
Set Obj = ADE.Get("MyObject")  'alternate equiv. form
```

API Errors: 41- "Invalid name for object"

OpenModel

Description: Reads a model from a disk file and opens it as the current model.

Parameters: FileSpec- string (the filename containing the model).

Return Value: ModelName – string (actual model name).

Usage:
```
modName = ADE.OpenModel ("C:\TMP\MYMODEL.ANA")
```

Remarks: Failure should be detected by checking whether the return value is "", not by checking for a zero ErrorCode. It is possible that some errors or warnings may occur during loading, and will thus be reflected in the ErrorCode, ErrorText, and OutputBuffer properties, even though the load was successful.

API Errors: 2– Warning (but load was successfully completed).
3– Lexical error (load was only partially successful)
4– Statement error (load was only partially successful)
39- "Model could not be found"

ReadScript(*filePath*)

Description: Reads an Analytica script file and executes it.

Parameters: ***filePath*** - string

Usage: `ADE.ReadScript ("C:\TMP\SCRIPT.MOD")`

API Errors: 39- "Script file could not be found"

ResetError

Description: Resets the error code, error text string associated with the error code, and the output buffer to default values. This function is normally used internally, but could be useful in other circumstances as well.

Usage `ADE.ResetError`

SaveModel(*filePath*)

Description: Saves the model to file ***filePath***

Parameters: ***filePath*** - string

Usage: `ADE.SaveModel("C:\TMP\CHANGES.ANA")`

API Errors: 46- "Model could not be saved"
49- "There is no model to save"

SaveModuleFile(*modName*, *filePath*)

Description: Saves module with identifier ***modName*** into file ***filePath***.

Parameters: ***modName*** – string
filePath - string

Return Value: boolean (success or failure)

Usage: `b = ADE.SaveModuleFile("Function_lib", "C:\TEMP\NEWMOD.ANA")`

API Errors: 41 – "Invalid name for object"
46 - "Module could not be saved"

Send

Description: Sends the string contained in the **Command** property as a command to be executed by ADE. See the *Analytica Scripting Guide* for details of commands and syntax.

Return Value: Boolean (success or failure)

API Errors: 1- "Unimplemented"
2- "Warning"
3- "Lexical error"
4- "Statement error"
5- "Expression error"
6- "Execution error"
7- "System error"
8- "Fatal error"
9- "Undefined variable error"
10- "Aborted"

SendCommand(*command*)

Description: Sends the string *command* property as a typescript command to be executed by ADE. See the *Analytica Scripting Guide* for details of commands and syntax.

SendCommand is a single method that is faster way to to execute a typescript command than using the **Send** method. Using Send requires two statements to execute a command:

```
ade.Command = "profile Val"
b = ade.Send( )
```

This can be done with a single SendCommand statement:

```
b = ade.SendCommand("profile Val")
```

Return Value: Boolean (success or failure)

API Errors:
1- "Unimplemented"
2- "Warning"
3- "Lexical error"
4- "Statement error"
5- "Expression error"
6- "Execution error"
7- "System error"
8- "Fatal error"
9- "Undefined variable error"
10- "Aborted"

Class CAObject

Properties

ClassType

Description: Contains the type of the Analytica object.

Data type: string

Access: read/write

Usage `classType = CAObject.ClassType`

Remarks: ADE currently supports the following types of Analytica objects: Decision, Chance, Constant, Index, Module, and Variable.

Name

Description: Contains the name given to the Analytica object.

Data type: string

Access: read/write

Usage: `CAObject.Name = "NewName"`

API Errors: 41- "Invalid name for object"

RenderingStyle

Description: Contains a **CARenderingStyle** object that controls how data is returned from Analytica. This property is inherited from the DefaultRenderingStyle property of **CAEngine** when the object is first instantiated. Its settings control how data is returned from **CAObject::Result** and **CAObject::GetAttribute**. Also, the setting will be inherited by any **CATable** created from the object by the Evaluate or ResultTable methods.

Data type: **CARenderingStyle**

Access: read/write

Usage: `obj.RenderingStyle.StringQuotes = 0`

ResultType

Description: Specifies the treatment of uncertainty in the value obtained using the Result or ResultTable properties:
0 - Mid value (default)
1 - Mean |
2 - Sample
3 – PDF
4 – CDF
5 – Statistics
6 – Confidence Bands

Data type: integer

Access: read/write

Usage: CAObject.ResultType = 1

Methods

CreateDefTable(indexList)

Description: Creates an input table object in the definition attribute of the specified Analytica object with dimension specified by the ***indexList***. The **IndexList** parameter must contain an array of identifiers of existing Index variables (identical in form to the **IndexNames** method of class **CATable**). The number of indexes in ***indexList*** determines the number of dimensions of the table. One index may be "Self", meaning that one of the dimensions will be indexed by the Indexvals attribute of this variable. should be one of the entries in the array. Initially, the input table object's array will be filled with null elements, which can be changed using the SetDataByElements and SetDataByLabels methods of the class **CATable**.

Parameters: ***indexList*** - array of strings

Return Value: boolean (success or failure)

Usage:
```
Var.CreateDefTable (IndexList)
```

API Errors: 25- "Subscripts cannot be accessed"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in table"
32- "Index object not found"

DefTable

Description: Gives object of class **CATable** containing the input table for an Analytica Variable — or **Nothing** if the variable was not defined as an input table.

Data type: **CATable**

Access: read/write

Usage:
```
Dim CATable As Object
Set CATable = CAObject.DefTable
```

API Errors: 34- "Definition table not found"

Evaluate(*expr*)

Description: Parses and evaluates an Analytica expression expr.

Parameters: expr - string

Return Value: **CATable**

Usage:
```
dim tab As CATable = _
obj.Evaluate("Sum(Revenue,Division)")
```

API Errors: 35- "Attribute could not be retrieved"

GetAttribute(*attribName*)

Description: Gets the value of Attribute ***attribName*** of the object.

Parameters: ***attribName*** - string

Return Value: variant

Usage:
```
X = obj.GetAttribute("definition")
```

API Errors: 35- "Attribute could not be retrieved"

PictureToFile(fileName, mimeType)

Description: Causes ADE to retreive**CAObject** object's picture, if any, and save it to file **fileName**, in the format specified by **mimeType**.

Return Value: Boolean (success)

Parameters: **fileName** : string
mimeType : string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png").

Usage:
```
obj = ade.Get("Pi1");
bSuccess = obj.PictureToFile( @"C:\Temp\myPict.jpg","image/jpeg")
```

PictureToStream(stream, mimeType)

Description: Causes ADE to retreive **CAObject** object's picture, if any, and send it to stream **stream**, in the format specified by **mimeType**.

Return Value: Boolean (success)

Parameters: **stream** : string
MimeType : string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png").

Usage:
```
dim outStream as MyStreamWrapper = _
    new MyStreamWrapper(Response.OutputStream)
bSuccess = obj.PictureToStream( outStream,"image/jpeg")
```

Result

Description: Gives the result value of the object as a single value (not a table). ADE will first evaluate the value if it has not already been evaluated. If the value is an array, it returns it as a string of comma-delimited elements.

Data type: variant

Access: read

Usage:
```
Dim x
X = CAObject.Result
```

API Errors: 37- "Could not retrieve result"

ResultTable

Description: Gives the value of the object as a **CATable**, with zero or more dimensions, or **Nothing** if the object cannot be evaluated. ADE first evaluates the variable if necessary.

Data type: **CATable**

Access: read

Usage:
```
Dim CATable As Object
Set CATable = CAObject.ResultTable
```

API Errors: 38- "Could not get result table"

SetAttribute(*attribName*, *value*)

Description: Sets Attribute ***attribName*** of the Object to ***value.*** It returns **true** if successful or **false** if not.

Parameters: ***attribName*** - string
value - Variant

Return Value: Boolean

Usage:
```
bool = obj.SetAttribute ("definition","A/B")
```

API Errors: 36- "Attribute could not be set"

Class CATable

Properties

GraphHeight

Description: Controls the height of the graph image returned by **GraphToStream** or **GraphToFile**.

Data type: integer (number of pixels)

Access: read/write

Usage: `tab.GraphHeight = NumDims`

GraphWidth

Description: Controls the width of the graph image returned by **GraphToStream** or **GraphToFile**.

Data type: integer (number of pixels)

Access: read/write

Usage: `x = CATable.TableType`

NumDims

Description: The number of dimensions of the table (zero if it is a scalar with no dimensions).

Data type: integer

Access: read

Usage: `x = CATable.NumDims`

RenderingStyle

Description: Contains a **CARenderingStyle** object that controls how atomic values are interpreted when transferred to and from table cells. Definition tables inherit this property from the **DefaultDefTableRenderingStyle** property of **CAEngine**. **Result** table inherits this property from the **CAObject** that created the table.

Data type: **CARenderingStyle**

Access: read/write

Usage: `tab.Renderingstyle.NumberAsText = true`

ResultType

Description: Contains the type of result that was computed, and controls the type of result computed if the table is updated. Possible value are the same as for **CAObject::ResultTable**.

Data type: integer

Access: read/write

Usage: `x = CATable.ResultType`

TableType

Description: This property holds the type of the table (“D” for a definition table, and “V” for a result table)

Data type: string

Access: read

Usage:
```
x = CATable.TableType
```

Methods

AtomicValue

Description: Retrieves the scalar value from a zero-dimensional **CATable** object. A zero-dimensional table results when a result is not an array, or when you call Slice or Subscript on a one-dimensional array.

Return Value: variant

Parameters: none

Usage:
```
x = tab.AtomicValue
```

GetDataByLabels(indexLabels)

Description: Retrieves the value of an input table cell according to **indexLabels**, which specify the label for each index of the table in order.

Return Value: variant

Parameters: values of indexes (*Variant*); the number of elements in*Variant* must be equal to **NumDims.**

Usage:
```
IndexLabs (1) = 3
IndexLabs (2) = "green"
W = Var.DefTable.GetDataByLabels (IndexLabs)
```

Or

```
IndexLabs (1) = 3
IndexLabs (2) = "green"
W = Var.ResultTable.GetDataByLabels (IndexLabs)
```

If the table has only one dimension, the parameter need not be an array:

```
W = Var.ResultTable.GetDataByLabels("green")
```

API Errors:
24- "Subscripts must be an array of variants"
25- "Subscripts cannot be accessed"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in table"
30- "Position does not exist"

GetDataByElements

Description: Retrieves the value of input table cell according to index values.

Return Value: variant

Parameters: index values (Variant), number of elements in the variant must be equal to **NumDims.**

Usage:
```
IndexPtrs (1) = 1
IndexPtrs (2) = 2
W = Var.DefTable.GetDataByElements (IndexPtrs)
```

Or

```
IndexPtrs (1) = 1
IndexPtrs (2) = 2
W = Var.ResultTable.GetDataByElements (IndexPtrs)
```

If the table is one dimensional, then an array is not needed:

```
W = Var.ResultTable.GetDataByElements (1)
```

API Errors:
24- "Subscripts must be an array of variants"
25- "Subscripts cannot be accessed"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in table"
29- "Position specified is out of bounds"
30- "Position does not exist"
31- "Illegal Position Specified In Table"

GetIndexObject

Description: Retrieves an index object by its name

Return Value: **CAIndex**

Parameters: index name: string

Usage:
```
dim AI as Object
Set AI = AObj.GetIndexObject (IndexName)
```

Remarks: If **ObjName** is not valid the method returns `Nothing`.

API Errors: 32- "Index object not found"

GetSafeArray

Description: Retrieves the **CATable** as a safe array (i.e., a Visual Basic array). The ordering of the dimensions is controlled by the **SetIndexOrder** method. The elements of each dimension are indexed 1 to N, where N is the size of each index.

Return Value: Array

Usage:
```
dim Var As Object
Dim curTable
curTable = Var.GetSafeArray
```

API Errors: None.

GraphToFile(fileName, mimeType)

Description: Creates a graph image of the data contained in the **CATable** object formatted using the ***mimeType*** and writes it to file ***fileName***. It uses attribute settings for the **CAObject** from which the **CATable** was obtained to control graph settings, uncertainty settings, and number format. The **GraphWidth** and **GraphHeight** properties control the size of the graph image in pixels.

Return Value: Boolean (success)

Parameters: **fileName** : string
mimeType : string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png").

Usage:
```
tab.GraphWidth = 350
tab.GraphHeight = 200
b = tab.GraphToFile( "C:\data\trends.jpg","image/bmp")
```

GraphToStream(stream, mimeType)

Description: Creates a graph image of the data contained in the **CATable** object formatted using the ***mimeType***, and writes it to *stream*, a Windows IStream. Note: An IStream is not interchangeable with a .NET System.Io.Stream object (including a Response.OutputStream object in ASP.NET). A wrapper class is necessary for converting between these. The size of the image in pixels is controlled by the **GraphWidth** and **GraphHeight** properties of the table. The graphing options are controlled by the

Return Value: Boolean (success)

Parameters: **stream** : string
MimeType : string (usually "image/jpeg", "image/bmp", "image/tiff", or "image/png").

Usage:
```
tab.GraphWidth = 350
tab.GraphHeight = 200
dim outStream as MyStreamWrapper = _
new MyStreamWrapper(Response.OutputStream)
b = tab.GraphToStream( outStream, "image/jpeg")
```

IndexName(IndexNumber)

Description: Takes one parameter, **IndexNumber**, and returns the name of the corresponding index for **CATable**.

Return Value: string

Parameters: **IndexNumber** : short int.

Usage:
```
dim string as indexTitle = Var.DefTable.IndexName (1)
```

IndexNames

Description: Returns the names of the indexes of the table as an array. Use indexes in this order when using the **GetDataBy** and **SetDataBy** methods to get or set elements of the table. You can change the order with the **SetIndexOrder** method.

Data type: string array with dimension from 1 to object's **NumDims** property

Access: read

Usage:
```
Dim names(k) of String = Var.DefTable.IndexNames (k)
```

PutSafeArray

Description: Replaces the current table represented by this object with another table of the same dimensions.

Return Value **CATable**

Parameters: the table (Visual Basic Array) that will replace the current table)

Usage:
```
Dim Var As Object
Dim TheArray
```

```
TheArray = Var.GetSafeArray
Var.PutSafeArray (TheArray)
```

API Errors: 24- "Subscripts must be an array of variants"
50- "Safe-array has incorrect size or number of dimensions"

SetDataByLabels

Description: Sets the value of an input table cell according to its index labels.

Return Value: boolean (success or failure)

Parameters: cell value (Variant), values of indexes (Variant), number of elements in this variant must be equal to **NumDims.**

Usage:

```
IndexVals (1) = 3
IndexVals (2) = 'green'
RetVal= Var.DefTable.SetDataByLabels (W, IndexVals)
```

If the table is one-dimensional, then an array is not needed:

```
W = Var.DefTable.SetDataByLabels (W, "green")
```

API Errors: 24- "Subscripts must be an array of variants"
25- "Subscripts cannot be accessed"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in table"
30- "Position does not exist"
31- "Illegal position specified in table"

SetDataByElements(*value*, *indexVals*)

Description: Sets the input table cell specified by ***indexVals*** to value. ***indexVals*** must contain a label for each index of the table, so the number of labels must equal ***NumDims***. If the table has just one dimension, indexVals may be the label for that one index rather than an array of labels.

Return Value: boolean (success or failure)

Parameters: value: Variant, indexVals: Variant

Usage:

```
IndexPtrs (1) = 1
IndexPtrs (2) = 2
RetVal = Var.DefTable.SetDataByElements (W, IndexPtrs)
```

If the table is one-dimensional, then an array is not needed:

```
W = Var.DefTable.SetDataByElements (W, 1)
```

API Errors: 24- "Subscripts must be an array of variants"
25- "Subscripts cannot be accessed"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in table"
29- "Position specified is out of bounds"
31- "Illegal position specified in table"
51- “Element position is non-numeric”

SetIndexOrder(*indexNames*)

Description: Sets the order of the indexes in the table to the order of ***indexNames***, which must contain the names of only and all the indexes of the table, assuming it has more than one index. This order determines the order used by **SetDataByElements**, **SetDataByLabels**, **GetDataByElements**, and **GetDataByLabels** to access a cell in a table.

Return Value: boolean (success or failure)

Parameters: ***indexNames****:* array of strings

Usage:
```
IndexVals (1) = 'X'
IndexVals (2) = 'Y'
RetVal = Var.DefTable.SetIndexOrder (W, IndexVals)
```

API Errors: 24- "Subscripts must be an array of variants"
26- "Lower bound of subscript array inaccessible"
27- "Upper bound of subscript array inaccessible"
28- "Must specify at least one element in array"
52- "Specified name is not an index of the array"

Slice(*indexObj*, *n*)

Description: Returnss the ***n***th slice of the table over index *indexObj.* The result is a new **CATable** object with one fewer dimensions than the table to which it is applied.

Parameters: **indexObj: CAObject**
n : Integer – the 1-based slice position along index.

Return Value: **CATable**

Usage:
```
dim In1 as CAIndex = tab.GetIndexObject("In1")
dim subTab as CATable = tab.Slice( In1, 1 )
```

Subscript(*indexObj*, *label*)

Description: Returns a slice of a table for which index ***indexObj*** is equal to ***label***. It retruns a new **CATable** object with one fewer dimensions than the original table.

Parameters: **indexObj: CAObject**
label : variant – the label in index.

Return Value: **CATable**

Usage:
```
dim In1 as CAIndex = tab.GetIndexObject("In1")
dim subTab as CATable = tab.Slice( In1, "SomeLabel" )
```

Update

Description: For Definition Tables: Updates an existing input table in the definition attribute of an Analytica object. Use this method after setting one or more **SetDataBy** methods to direct the API to send the new table data to the Analytica Decision Engine Server. For Result Tables: Retrieves an updated version of the result table from the Analytica Decision Engine Server.

Return Value: Boolean (success or failure)

Usage:
```
Var.DefTable.Update
```

Remarks: Use the **CreateDefTable** method to replace the current definition attribute of an Analytica object with an input table.

API Errors: None.

Class CAIndex

Properties

IndexElements

Description: Returns the number of elements in the index.

Data type: integer

Access: read

Usage: `x = CAIndex.IndexElements`

Name

Description: Contains the name given to the Analytica index.

Data type: string

Access: read/write

Usage: `theName = CAIndex.Name`

API Errors: 41- "Invalid name for object"

RenderingStyle

Description: Contains a **CARenderingStyle** object that controls how values are returned from GetValueByNumber.

Data type: **CARenderingStyle**

Access: read/write

Usage: `theIndex.RenderingStyle.StringQuotes = 1`

Methods:

GetNumberByValue

Description: returns the position of an index label in an index.

Parameters: **Value** - variant

Return Value: integer

Usage: `n = Anindex.GetNumberByValue (Value)`

API Errors: 22- "Value not found in index"

GetValueByNumber

Description: returns the index label at the specified position in the index.

Parameters: **Number** - integer

Return Value: variant

Usage: `W = AnIndex.GetValueByNumber (Number)`

API Errors: 23- " Illegal position in index"

Class CARenderingStyle

Properties

GeneralExpression

Description: Determines how string values are interpreted when they are written to a definition table. When True (default), a string is taken to be an expression, which must parse to be a valid expression. When False, a string value is stored as a literal string. For example, the value “Pi” would be interpreted as the identifier *Pi* when GeneralExpression is true, and would thus evaluate to 3.141592654, but the same string would be interpreted as a literal character string when GeneralExpression is false, and would evaluate to two-character textual string.

Data type: Boolean

Access: read / write

Usage:
```
deftab.RenderingStyle.GeneralExpression = false
```

FullPrecision

Description: When True a numeric values rendered as text (see the **NumberAsText** property will have the maximum number digits needed to represent the number at full precision (usually 16). If False, it so that a loss of precision does not occur.

Data type: Boolean

Access: read / write

Usage:
```
dim res As CATable = obj.Evaluate("sqrt(2)")
res.RenderingStyle.NumberAsText = true
res.RenderingStyle.FullPrecision = false
dim s As String = res.AtomicValue  ' returns "1.414"
res.renderingStyle.FullPrecision = true
s = res.AtomicValue  ' returns "1.4142135623731"
```

NumberAsText

Description: Controls whether numeric results or numeric table cell definitions are returned as floating point numbers or as formatted strings. When true, the number format for the current object will control how the number is formatted as a string (except that the **FullPrecision** property may override the number of digits in the number format).

Data type: Boolean

Access: read / write

Usage:
```
deftab.RenderingStyle.NumberAsText = true
```

ReferenceAsText

Description: Some Analyica expressions may evaluate structured value (such as a tree) containing references. This property controls whether references are returned as **CATable** objects (containing the de-referenced value), or rendered as text. By default, they are returned as **CATable** objects. Note that in Analytica, a reference is treated as atomic, even though its dereferenced value may be array valued.

Data type: Boolean

Access: read / write

Usage:
```
obj.SetAttribute("definition", "\(A^t)")
def res as CATable = obj.Evaluate("\(A^t)")
res.RenderingStyle.ReferencesAsText = false
def derefdVal as CATable = res.AtomicValue
```

SafeArrayLowerBound

Description: The lower bound for safe arrays returned by **CATable::GetSafeArray.** Default is 1.

Data type: Integer

Access: read / write

Usage:
```
ADE.DefaultRenderingStyle.SafeArrayLowerBound = 0
```

StringQuotes

Description: Controls whether textual values are returned with explicit quotation marks surrounding a string, according to its value:
0 = no quotes around strings— *e.g.,* 0.25
1 = single quotes — *e.g.*, '0.25'
2 = double quotes — *e.g.,* "0.25"

When **NumberAsText** is true, with no quotation marks around string values, the numeric value 0.25 and the string containing the four characters "0.25" would be indistinguishable. If you want to be able to distinguish them, set the value of **StringQuotes** to 1 or 2.

Data type: Integer:
0 = no quotes,
1 = 'single quotes',
2 = "double quotes".

Access: read / write

Usage:
```
deftab.RenderingStyle.StringQuotes = 2
```

UndefValue

Description: This property specifies the value returned when the Analytica value is Undefined. By default, ADE returns the special value Variant **Empty** — for example, when GetAttribute is applied to an Attribute that was not set. Some scripting languages cannot manipulate the value **Empty**: In this case you can set **UndefValue** to something more convenient, such as **Null** or the empty string.

Data type: Variant

Access: read / write

Usage:
```
ADE.DefaultRenderingStyle.UndefValue = Null
```

VarTermFormat

Description: Controls how a varTerm (a pointer to an Analytica object) is returned. A varTerm may occur in a definition cell of a table if that definition consists of a single identifier. There are also some rare Analytica expressions that can produce a VarTerm in a result. If **VarTermFormat** = 0, it will return the identifier of the Analytica object as a string; if 1, a **CAObject** for the Analytica object; or if 2, the Title of the object as a string.

Data type: Integer:
0 = identifier,
1 = **CAObject**,
2 = title

Access: read / write

Usage: `deftab.RenderingStyle.VarTermFormat = 1`

Appendices

The following appendices provide you with:

- The complete list of ADE error messages
- A glossary of Analytica terms

Appendix A. Error Codes

ADE Error Codes

Error Code	Meaning/Error Text
0	"OK"
1	"Unimplemented"
2	"Warning"
3	"Lexical error"
4	"Statement error"
5	"Expression error"
6	"Execution error"
7	"System error"
8	"Fatal error"
9	"Undefined variable error"
10	"Aborted"
11	"Analytica expired -- contact Lumina"
12	*not currently used*
13	*not currently used*
14	*not currently used*
15	*not currently used*
16	*not currently used*
17	*not currently used*
18	
19	*not currently used*
20	"Analytica is uninitialized"

API Error Codes

21	"Bad parameter passed"
22	"Value not found in index"
23	"Illegal position in index"
24	"Subscripts must be an array of variants"

25	"Subscript cannot be accessed"
26	"Lower bound of subscript array inaccessible"
27	"Upper bound of subscript array inaccessible"
28	"Must specify at least one element in table"
29	"Position specified is out of bounds"
30	"Position does not exist"
31	"Illegal position specified in table"
32	"Index object not found"
33	"Illegal index number specified"
34	"Definition table not found"
35	"Attribute could not be retrieved"
36	"Attribute could not be set"
37	"Could not retrieve result"
38	"Could not get result table"
39	"Module/Model/Script could not be found"
40	"Object could not be created"
41	"Invalid name for object"
42	"Object name already in use"
43	"Current module could not be retrieved"
44	"Module could not be set"
45	"Model could not be created"
46	"Model/Module could not be saved"
47	"Illegal command"
48	"Invalid object class"
49	"There is no model in memory to save"
50	"Safe-array has incorrect size or number of dimensions"
51	"Element position is non-numeric"
52	"Specified name is not an index of the array"
53	"Insufficient memory"
54	"Unrecognized MIME type"
55	"Value is not atomic"
56	"Operation allowed only on a result CATable, not on a definition table"
57	"First param is not an IStream*"
58	"Subscript array contains the wrong number of elements There should be one element for each dimension"
59	"CATable is not associated with an object, so it cannot be updated"
60	"A result table is read-only"

61	"Expression could not be parsed"
62	"Error evaluating expression"
63	"GraphWidth and GraphHeight must be positive"
64	"Value is atomic (not an array) To get value, use AtomicValue method"
65	"Index value could not be computed"
66	"No picture stored with object"
67	"Internal picture format not supported"
68	"Filename too long"
69	"Result cannot be graphed"
70	"Definition is Hidden"

Appendix B. Glossary

ADE	See “Analytica Decision Engine.”
Alias	A node in a diagram that refers to a Variable or other node located somewhere else, usually in another module. An alias permits you to display a Variable in more than one module. An alias node is distinguished by having its title in italics.
Analytica Browser	A free edition of Analytica that allows a user to evaluate and view results, and change input fields; however, from Analytica Browser a user cannot enter edit mode or otherwise change the content of a model. Copies of Analytica without a valid registration number run as the Analytica Browser.
Analytica Decision Engine	A product sold by Lumina Decision Systems, Inc., separate from Analytica. With the Analytica Decision Engine (ADE), you embed the Analytica computation engine in your web-server backend or in your custom applications built in Visual Basic, C++, Microsoft Office, or any language supporting ActiveX Automation or COM.
Analytica Enterprise	A edition of Analytica for users who intend to share data or models with others in their organization. Analytica Enterprise contains all features of Analytica Pro as well as functions for accessing ODBC databases and features for protecting your intellectual property.
AnalyticaProfessional edition	The standard fully-functional edition of Analytica. Analytica Pro provides all the features and functionality required to create, edit, and evaluate models.
Analytica Trial	A fully-functional, but expiring, edition of Analytica. Analytica Trial can be downloaded from the Lumina web site (www.lumina.com) for those wishing to "test drive" the product. Analytica Trial contains the complete functionality of Analytica Pro. After expiration, Analytica Trial converts to Analytica Browser.
Array	A collection of values that can be viewed as one or more tables. An array has one or more dimensions; each dimension is identified by an index.
Array abstraction	See “Intelligent Array Abstraction™.”
Arrow	An arrow or influence from one Variable node to another indicates that the origin node affects (influences) the destination node. If the nodes depict Variables, the origin Variable usually appears in the definition of the destination Variable.
Arrow tool	The Arrow tool, or Influence Arrow tool, is in the shape of a left-to-right pointing arrow cursor. The Arrow tool is used to draw arrows connecting Variables to create relations between them.
Attribute	A property or descriptor of an Object, such as its title, description, definition, value, or inputs.
Attribute panel	An auxiliary window pane that you can open below a diagram or outline window. Use the Attribute panel to rapidly examine one Attribute at a time of any Variable in the model, by selecting the Variable and then the Attribute from a popup menu.
Author	An Attribute recording the names of the person or people who created the model, or other Object.
Behavior analysis	Model behavior analysis is a type of sensitivity analysis in which you specify a set of alternative values for one or more inputs and examine the effect on selected model output Variables. It is also known as parametric analysis.
Browse-only models	Analytica Enterprise users can save a copy of their model in a browse-only form. When a browse-only model is loaded into any edition of Analytica, the user cannot enter edit mode, and therefore can only make changes to Variables with input nodes. Browse-only models are also obfuscated.

Browse tool The Browse tool is in the shape of a hand. With the Browse tool, you can examine the diagram but cannot make any changes, except to change the values in input nodes.

Chance Variable A Chance Variable is uncertain and cannot be directly controlled by the decision maker. Usually, it is defined by a probability distribution. A Chance Variable is depicted as an oval node.

Check The check Attribute contains an expression that checks the validity of the value of a Variable. It displays a message when the Variable's value is out of specified bounds.

Class The type of Analytica Object: decision, chance, objective, or index Variable; function; module; library; form; model.

Cloaking See “Definition Hiding.”

Conditional dependency A Chance Variable *a* is conditionally dependent on another Variable *b* if the probability of a value of *a* depends on the value of *b*. If *a* is defined by a probability table, *b* may be an index of its probability table.

Constant A Variable whose value is not probabilistic, and does not depend on other Variables, such as the number of minutes in an hour.

Continuous distribution A probability distribution defined for a continuous Variable—that is, for a real-valued Variable. Example continuous distributions are beta, normal, and uniform. Compare to “Discrete distribution.”

Continuous Variable A Variable whose value is a real number—that is, one of an infinite number of possible values. Its range can be bounded (for example, between 0 and 1) or unbounded. Compare to “Discrete Variable.”

Created The date and time at which the model was first created. This model Attribute is entered automatically, and is not user-modifiable.

Cumulative probability distribution A representation of a probability distribution that plots the cumulative probability that the actual value of the uncertain Variable *x* will be less than or equal to each possible value of *x*. The cumulative probability distribution is a display option in the Uncertainty View popup menu.

Decision Variable A Variable that the decision maker can control directly. Decision Variables are represented by rectangular nodes.

Definition A formula that defines how to compute a Variable’s value. It can be a simple number, a mathematical expression, a list of values, a table, or a probability distribution. In text format, it is limited in length to 32,000 characters.

Definition Hiding A feature in Analytica Enterprise for protecting your intellectual property when distributing models you have created to others. Definition hiding controls whether the end-user of your model can view the definitions of selected nodes.

Description Text explaining what the node represents in the real system being modeled. It is limited in length to 32,000 characters.

Deterministic table A deterministic function that gives the value of a Variable *x* conditional on the values of its input Variables. The input must all be discrete Variables. The table is indexed by each of its inputs, and gives the value of *x* that corresponds to each combination of values of its inputs.

Deterministic value A Variable's deterministic value, or mid value, is a calculation of the Variable's value assuming all uncertain inputs are fixed at their median values.

Deterministic (determ) Variable A Variable that is a deterministic function of its inputs. Its definition does not contain a probability distribution. The value of a deterministic Variable can be probabilistic if one

or more of its inputs are uncertain. A deterministic Variable is displayed as a double oval. You can also use a general Variable (rounded rectangle) to depict a deterministic Variable.

Determtable See “Deterministic table.”

Diagram See “Influence diagram.”

Dimension An array has one or more dimensions. Each dimension is identified by an index Variable. When an array is shown as a table, the row header (vertical) and column headers(horizontal) give the two dimensions of the table.

Discrete distribution A probability distribution over a finite number of possible values. Example discrete distributions are Bernoulli and the Probtable function. Compare to “Continuous distribution.”

Discrete Variable A Variable whose value is one of a finite number of possible values. Examples are the number of days in a month (28, 29, 30, or 31), or a Boolean Variable with possible values `True` and `False`. A Variable that is defined as a list or list of labels is discrete. Compare to “Continuous Variable.”

Domain The possible outcomes of a Variable. The Domain has a type as well as value. The possible types are List of labels, List of numbers, or Continuous; the default type is Continuous, except for Variables defined with the `Choice()`, `Probtable()`, and `Determtable()` functions.

Dynamic Variable A Variable that depends on the system Variable *Time* and is defined by the `Dynamic()` function. A dynamic Variable can depend on itself at a previous time period, directly or indirectly, through other dynamic Variables.

Edit Table A definition that is a table is also called an Edit Table because it can be edited.

Edit tool The Edit tool is in the shape of the normal mouse pointer cursor. The Edit tool is used to create a new model or to change an existing model. It allows you to move, resize, and edit nodes, and exposes the Arrow tool and node palette.

Excel Graph The graphing engine of Microsoft Excel®. Users who have Excel installed on their computers can take advantage of Excel Graph to graph results.

Expression A formula that can contain numbers, Variables, functions, distributions, and operators, such as `0.5`, `a-b`, or `Min(x)`, combined according to the Analytica language syntax. The definition of a Variable must contain an expression.

Expression type The Expression popup menu, which appears above the definition field, allows you to change the definition of a Variable to one of several different kinds of expressions. Expression types include expression, list (of expressions or numbers), list of labels (text values), table, probability table, and distribution. Any definition, regardless of expression type, can be viewed as an expression.

File Info The name of the file and folders in which the model was last saved.

Filed library A library whose contents are saved in a file separate from the model that contains it. A filed library can be shared among several models without making a copy for each model.

Filed module A module whose contents are saved in a file separate from the model that contains it. A filed module can be shared among several models without making a copy for each model.

Fractile The median is the 0.5 fractile. More generally, there is probability *p* that the value is less than or equal to the *p* fractile. Quantile is a synonym for fractile. (Fractal is something different!) Compare to “Percentile.”

General Variable A Variable that can be certain or probabilistic. It is often convenient to define a Variable as a general Variable without worrying about what particular kind of Variable it is. A general Variable is depicted by a rounded rectangle node.

Identifier A short name for a Variable used in mathematical expressions in definitions. An identifier must start with a letter, have no more than 20 characters, and contain only letters, numbers, and '_' (underscore, used instead of a space). Each identifier in a model must be unique. Compare to "Title."

Importance analysis Importance analysis lets you determine how much effect the uncertainty of one or more input Variables has on the uncertainty of an output Variable. Analytica defines importance as the rank order correlation between the sample of output values and the sample for each uncertain input. It is a robust measure of the uncertain contribution because it is insensitive to extreme values and skewed distributions.

Unlike commonly used deterministic measures of sensitivity, this rank order correlation averages over the entire joint probability distribution. Therefore, it works well even for models where the sensitivity to one input depends strongly on the value of another.

Index An index of an array identifies a dimension of that array. An index is usually a Variable defined as a list, list of labels, or sequence. An index is often, but not always, a Variable with a node class of Index.

Indexes Plural of index. Indicates a set of index Variables that define the dimensions of a table (in an Edit Table or value).

Index selection area The top portion of a Result window, containing a description of the result and other information about the dimensions of the result.

Index Variable A class of Variable, defined as a list, list of labels, or sequence, that identifies the dimensions of an array—for example, in an Edit Table. An Index Variable is depicted as a parallelogram node. Variables of other classes whose definition or domain consist of list, list of labels, or sequence can also be used to identify the dimensions of an array, and are sometimes referred to as index Variables.

Influence arrow See "Arrow."

Influence cycle A cyclic dependency occurs when a Variable depends on itself directly or indirectly so that the arrows form a directed circular path. The only cyclic dependencies allowed in Analytica are in Variables using the `Dynamic()` function that contain a time lag on the cycle.

Influence diagram An intuitive graphical view of the structure of a model, consisting of nodes and arrows. Influence diagrams provide a clear visual way to express uncertain knowledge about the state of the world, decisions, objectives, and their interrelationships.

Innermost dimension The dimension of an array that varies most rapidly in the `Table()` function. The innermost dimension is the last index listed in a `Table()` or `Array()` function. Compare to "Outermost dimension."

Input A Variable that appears in the definition of the selected Variable. See also "Output."

Input arrowhead An arrowhead pointing into a node, indicating that the node has one or more inputs from outside its module. Click on the arrowhead for a popup menu of the input Variables.

Inputs A list of the Variables or functions on which this Variable or function depends. The inputs are determined by the arrows drawn to and the Variables or functions referred to in this Variable's or function's definition or check Attribute. See also "Outputs."

Intelligent Array Abstraction™ A powerful key feature of the Analytica Engine that automatically propagates and manages the dimensionality of multidimensional arrays within models.

Key In a results graph, the key shows the value of the key index Variable that corresponds to each curve, indicated by pattern or color.

Kurtosis A measure of the peakedness of a distribution. A distribution with long thin tails has a positive kurtosis. A distribution with short tails and high shoulders, such as the uniform distribution, has a negative kurtosis. A normal distribution has zero kurtosis.

Last Saved The date and time at which the model was last saved. This model Attribute is entered automatically, and is not user-modifiable. If the model is new, this field remains empty until the model is first saved.

Library A model component that typically contains a collection of user-defined functions and/or Variables to be shared.

List A type of expression available in the Expression popup menu consisting of an ordered set of numbers or expressions. A list is often used to define Index and Decision Variables.

List of labels A type of expression available in the Expression popup menu consisting of an ordered set of text items. A list of labels is often used to define Index and Decision Variables.

Matrix A two-dimensional array of numbers with indexes of equal length.

Mean The average of the population, weighted by the probability mass or density for each value. The mean is also called the ***expected value***. The mean is the center of gravity of the probability density function.

Median The value that divides the range of possible values of a quantity into two equally probable parts. Thus, there is 0.5 probability that the uncertain quantity is less than or equal to the median, and 0.5 probability that it is greater than the median.

Mid value The result of evaluating a Variable deterministically, holding probability distributions at their median value. Analytica calculates the mid value of a Variable by using the mid value of each input. The mid value is a measure of central value, computed very quickly compared to uncertainty values. Compare “Probvalue.”

Mode The most probable value of the quantity. The mode is at the highest peak of the probability density function. On the cumulative probability distribution, the mode is at the steepest slope, at the point of inflection.

Model A module, or a hierarchy of linked and/or embedded modules and libraries, on which you work during an Analytica session; the main, or root, module at the top of the module hierarchy. Between sessions, a model is stored in an Analytica document file.

Module A collection of related nodes, typically including Variables, functions, and other modules, organized as a separate Influence Diagram. A module is depicted in a diagram as a node with a thick outline.

Module hierarchy A model can contain several modules, each one containing details of the model. Each module can contain further modules, containing still more detail. This module hierarchy is organized as a tree with the model at the top. You can view the hierarchical structure in the Outline window.

Multimodal distribution A probability distribution that has more than one mode.

Node A shape, such as a rectangle, oval, or hexagon, that represents an Object in an Influence Diagram. Different node shapes are used to represent different types of Variables.

Obfuscated Saved in a non-human-readable (i.e, encrypted) form. Obfuscation provides a mechanism for protecting intellectual property. Analytica Enterprise users can distribute

obfuscated copies of their models to their end-users. In Analytica, obfuscation also has the effect of making settings for definition hiding and/or browse-only mode permanent.

Object A Variable, function, or module in an Analytica model. Each Object is depicted as a node in an Influence Diagram and is described by a set of attributes. See also “Class,” “Node,” “Attribute,” and “Influence diagram.”

Object Finder A dialog box used to browse and edit the functions and Variables available in a model.

Object window A view of the detailed information about a node. The Object window shows the visible attributes, such as a node’s type, identifier, and description.

Objective Variable A Variable that evaluates the overall desirability of possible outcomes. The objective can be measured as cost, value, or utility. A purpose of most decision models is to find the decision or decisions that optimize the objective—for example, minimizing cost or maximizing expected utility. An objective Variable is represented by a hexagonal node.

OLE Linking A standard in the Windows operating system for sharing data between applications.

Operator A symbol, such as a plus sign (+), that represents a computational process or action such as addition or comparison.

Outermost dimension The dimension of an array that varies least rapidly in the `Table()` function. The outermost dimension is the first index listed in a `Table()` or `Array()` function. Compare to “Innermost dimension.”

Outline window A view of a model that lists the objects it contains as a hierarchical outline.

Output A Variable whose definition refers to the selected Variable. See also “Input.”

Output Arrowhead An arrowhead pointing out of a node, indicating that the node has one or more outputs outside its module. Click on the arrowhead for a popup menu of the output Variables.

Outputs A list of the Variables or functions that depend on this Variable or function. The outputs are determined by the arrows drawn from this Variable or function and the Variables or functions in whose definition or check Attribute this Variable or function appears. See also “Inputs.”

Parameters The arguments of a function.

Parametric analysis See “Behavior analysis.”

Parent diagram The diagram for the module that contains this Object.

Percentile The median is the fiftieth percentile (also written as 50%ile). More generally, there is probability *p* that the value is less than or equal to the *pth* percentile. Compare to “Fractile.”

Probabilistic Variable A Variable that is uncertain, and is described by a probability distribution. A probabilistic Variable is evaluated using simulation; its result is an array of sample values indexed by *Run*.

Probability bands Probability bands are a way to display the uncertainty in a value by showing percentiles from its distribution—for example, the 5%, 25%, 50%, 75%, and 95% percentiles. On a graph, these often appear as bands around the median (50%) line. Probability bands are also referred to as credible intervals.

Probability density function A representation of a probability distribution that plots the probability density against the value of the Variable. The probability density at each value of *X* is the relative probability that *X* will be at or near that value. The probability density function can be displayed for continuous, but not discrete Variables. It is a display option in the Uncertainty View popup menu. Compare to “Probability mass function,” which is used with discrete Variables.

Probability distribution A probability distribution describes the relative likelihood of a Variable having different possible values.

Probability mass function A probability mass function is a representation of a probability distribution for a discrete Variable as a bar graph, showing the probability that the Variable will take each possible value. The probability mass function can be displayed for discrete, but not continuous Variables. It is a display option in the Uncertainty mode View menu. Compare to "Probability density function," which is used with continuous Variables.

Probability table A table for specifying a discrete probability distribution for a Chance Variable. In a probability table, you specify the numerical probability for each value in the domain of *the Variable*. If *the Variable* depends on (that is, is conditioned by) other discrete Variables, each of these conditioning Variables gives an additional dimension to the table, so you can specify the probability distribution conditional on the value of each conditioning Variable.

Probtable See "Probability table."

Probvalue The probabilistic value of a Variable, represented as a random sample of values from the probability distribution for the Variable. The probvalue for a Variable is based on the probvalue for the inputs to the Variable. See also "Probabilistic Variable" and compare to "Mid value."

Reducing function A function that operates on an array over one of its indexes. The result of a reducing function has that dimension removed, and hence has one fewer dimension.

Remote Variable A Variable in another module, not shown in the active diagram. Typically a remote Variable is an input or output of a node in the active diagram.

Result view A window that shows the value of a Variable as a table or graph.

Sample An array of values selected at random from the underlying probability distribution for a quantity. Analytica represents uncertainty about a quantity as a sample, and estimates statistics, probability density function, and other representations of a probability distribution from the sample.

Sampling method A method used to generate a random sample from the probability distributions in a model (for example, Monte Carlo and Latin hypercube).

Scalar A value that is a single number.

Scatter plot A graph that plots the samples of two probabilistic Variables against each other.

Self A keyword used in two different ways:

- Refers to the index of a table that is indexed by itself. `Self` refers to the alternative values of the Variable defined by the table.
- Refers to the Variable itself, as a substitute for the Variable's identifier, in a Check Attribute expression or a `Dynamic` expression.

Sensitivity analysis A method to identify and compare the effects of various input Variables to a model on a selected output. Example methods for sensitivity analysis are importance analysis and model behavior analysis.

Skewness A measure of the asymmetry of the distribution. A positively skewed distribution has a thicker upper tail than lower tail, while a negatively skewed distribution has a thicker lower tail than upper tail. A normal distribution has a skewness of zero.

Slice A slice of an array is an element or subarray selected along a specified dimension. A slice has one less dimension than the array from which it is sliced.

Standard deviation The square root of the variance. The standard deviation of an uncertainty distribution reflects the amount of spread or dispersion in the distribution.

Suffix Numbers such as 10K, 123M, or 1.23u are in suffix notation. The suffix letter denotes a power of ten; for example, K, M, and u denote 10^3, 10^6, and 10^{-6}, respectively.

Symmetrical distribution A distribution, such as a normal distribution, that is symmetrical about its mean.

System function A function available in the Analytica modeling language. See also “User-defined function.”

System Variable A Variable that is part of the Analytica modeling language, such as *Samplesize* or *Time*.

Table A two-dimensional view of an array. The array can have more than two dimensions, but only two can be seen at one time. In the Result window, click on the **Table** button to select the table view of an array-valued result.

Tail The upper and lower tails of a probability distribution contain the extreme high and low quantity, respectively. Typically, the lower and upper tails include the lower and upper ten percent of the probability, respectively.

Title The full name of an Analytica Object. A Variable's or module's title is displayed in its node, in window titles, and in Object lists. It is limited to 255 characters. It can contain any characters, including spaces and punctuation. Compare to “Identifier.”

Uncertain value See “Probvalue.”

Units The units of measurement for a Variable. Units are used to annotate tables and graphs; they are not used in any calculation.

User-defined function A function that the user defines to augment the functions provided as part of the Analytica modeling language.

Value See “Mid value.”

Variable An Object that has a value, which may be text, a number, or an array.

Variance A measure of the uncertainty or dispersion of a distribution. The wider the distribution, the greater its variance.