A SYNOPSIS OF SOFTWARE TECHNOLOGIES USED IN TODAY'S ENGINEERING SOFTWARE

Ashley Hull, Ph.D.
Hasmet Genceli, Ph.D.
Michael W. Hlavinka, Ph.D., P.E.

Bryan Research & Engineering, Inc.
P.O. Box 4747
Bryan, TX 77805 USA
www.bre.com

ABSTRACT

Most engineers have little familiarity with the software technologies that provide the framework for the variety of applications they employ, from word processors to process simulators. While adequate for casual use of an application, a more thorough understanding of these technologies is required in order to extend an application and provide custom behavior. Usually the effort required to perform small customizations is not significant provided the framework is understood. This paper introduces Object-Oriented Programming (OOP), OLE Automation, and Extensible Markup Language (XML), three common technologies used in programs. A conceptual discussion of OOP is presented along with examples where the paradigm may be encountered. Automation is introduced with illustrations of how this feature can extend an application. Finally, XML is summarized along with a discussion of some of the tools and supporting technologies used with XML. The aim of this paper is to give a basic understanding of how and when these technologies can be exploited based on specific applications and tasks.

A SYNOPSIS OF SOFTWARE TECHNOLOGIES USED IN TODAY'S ENGINEERING SOFTWARE

INTRODUCTION

Commercial software today is normally built upon several software technologies. These technologies provide a common framework that is consistent from one application to another. This framework is present in applications from word processors to process simulators. While an understanding of these technologies is not required for casual use of the application, some knowledge of this framework will be required to extend an application to provide custom behavior outside of the design of the vendor.

This paper presents an overview of Object-Oriented Programming (OOP), OLE (Object Linking and Embedding) Automation, and Extensible Markup Language (XML). Object-Oriented Programming and XML are platform independent technologies while OLE Automation is specific to the Microsoft® Windows® platform. OLE Automation is presented in this paper because of its prevalent status in most types of applications used by engineers. Object-oriented programming is the framework upon which most technologies are based, including OLE and XML. OLE Automation is a programming technology that allows a user to extend an application by manipulating and accessing the objects within the program. XML is a text based user (or application) defined data presentation technology which can be used to exchange information between applications, or can be transformed into other presentations of the data using standardized techniques. XML initially found its use in data driven web based applications, and its definition is standardized by the World Wide Web Consortium (W3C). Most database applications support or utilize XML in some fashion. A user can also access the elements of an XML file by writing a program based on the object-oriented programming model defined by the XML standard.

For small tasks, extending an application to provide simple additional custom behavior is usually not very difficult to accomplish. Many have already done this. For example, to create a macro to perform some custom action in Microsoft Excel, Visual Basic for Applications (VBA) code is usually written to access the objects defined within Excel. This may be done directly using the programming language or indirectly using the macro recorder present in Excel. The syntax and other details of the language used to extend the program are usually not the primary obstacles to accomplishing this task. The primary obstacle usually is comprehending the object-oriented approach and its presentation in the context of the object model of Excel.

Our goal in this paper is to provide the reader with a fundamental background of these technologies in a concise manner. Obviously, a full explanation of the implementation of these technologies is not practical here. As you read the paper, do not concern yourself greatly with the syntax and other details. Rather concern yourself with the fundamental concepts presented. With this in mind, we hope you can determine if any of these technologies has application in your work with these programs. If so, other references may be required to perform the implementation. We will try to address the topics for an engineering audience, not a computer programming audience. Although our background is in the development of commercial process simulation software, we will strive to introduce the material in a generic manner that does not require extensive knowledge of process simulation.

OBJECT-ORIENTED PROGRAMMING

Virtually all software today, not only engineering software, is designed based on an Object-Oriented Programming (OOP) paradigm. Most programs including process simulators, word processors, spreadsheets, database applications, and e-mail programs are object-oriented. With such a wide variety of software being object-oriented, there is no doubt that the concept can be somewhat confusing to software users. Understanding the basics of object-oriented design is a prerequisite for extending an application to provide some user defined behavior. In this section, a definition of an object is provided in addition to its relationship to the user of an application.

An object-oriented programming language is a programming language that supports the precepts of object-oriented programming [1]. Today platform independent languages exist providing various benefits including C++, Java, and Python, in addition to languages specifically for the Microsoft Windows platform including Visual Basic and C#. The object-oriented concepts are the same in all of the languages; only the syntax and presentation are different. Object-oriented approaches to commercial software have been prevalent for more than a decade. Sadly, most undergraduate and graduate engineering education still primarily focuses on procedural based implementation using languages such as FORTRAN, which are usually no longer practical for development of modern large-scale applications and their extension by the end user. Switching from a procedural to an object-oriented paradigm typically requires some effort since the thought process is different.

Smith and Tockey [2] suggest that an object presents an individual, identifiable item, unit, or entity, either real or abstract with a well-defined role in the problem domain. As humans, we typically restrict our view of objects to tangible items that we can visualize or manipulate [3]. However, in computer applications, objects are much broader than simply physical, tangible objects. Many objects are abstract items, which work in conjunction with other abstract or tangible objects to provide the anticipated behavior of the application. Software objects that have a physical identity would include unit operation (pump, separator, etc.) and stream objects in process simulators, and page objects representing the content of a printed page in a drawing program or word processor. Abstract software objects include objects providing thermodynamic and physical property calculation as well as numerical method implementation. Essentially a program object has state, behavior, and identity [3].

Figure 1 presents the hierarchy of objects that could be present in a simplified drawing application. The figure shows that objects are typically present in object-oriented applications to serve as containers or collections of other objects. In Figure 1, the collections include the Pages collection, which contain all the pages of the document, and the Shapes collection, which contain the shapes on each page of the document. Therefore, the number of Page and Shape objects that represent individual items will generally be greater than one. In many instances, the collections are implemented as vectors of objects. In this presentation and in many applications, container objects have plural names while the contained object has a singular name.

As seen in Figure 1, objects generally have relation to other objects in an application. Frequently, this relationship is a parent, child, or sibling type relationship. Here, the page would be a child of the drawing object, and each shape a child of the page on which it resides. In a multi-page drawing, each page would have other sibling pages, and each shape on a page would generally have other sibling shapes. Often the parent-child-sibling relationship is a natural progression of what the application physically represents. In order to allow the user to better understand the relationship between objects in the application, many programs include a complete object model hierarchy diagram as part of their product documentation, such as presented in Figure 1.

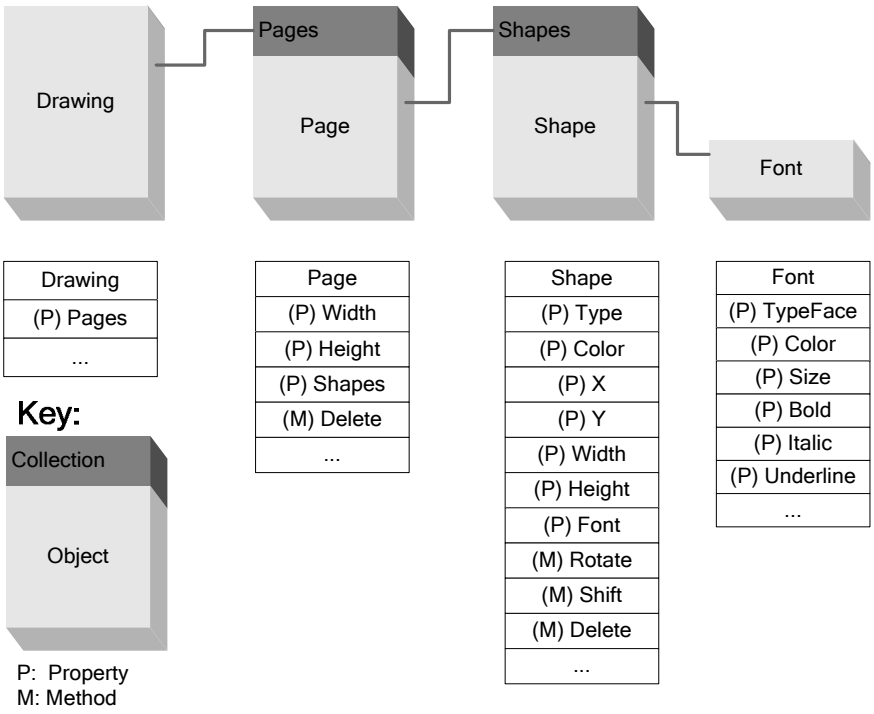

Figure 1. Object Model Hierarchy for a Simplified Drawing Program.

In order to customize or extend the behavior of an application, the user of the application will typically write macros, subroutines or functions to perform what is desired, and communicate with the underlying application through its object hierarchy or model. The application that hosts the object is termed a *server*. The end user subroutines and functions are termed a *client* of the object as they access and manipulate the state, behavior, and identity of the object. Each object in the server exposes its state, behavior, and identity through various public interfaces defined for the object. The client of an object connects to these interfaces and calls functions and subroutines defined by the interface. Each of the function and subroutine calls in the interface will generally have an associated argument list defined by the object. As with objects themselves, interfaces may be abstract definitions that provide a common set of routines exposed by more than one type of object in an application.

An interface typically defines two types of functions or subroutines. The first type of function defines and returns the state of the object. These functions are usually called *properties*. In Figure 1, property routines are prefaced with a (P) before the name of the routine (e.g., Width property of a Page). Property functions frequently return child objects rather than simple values. Examples of this type of property would include the Pages and Shapes collections of the Drawing and Page object, respectively, and the Font property of the Shape object in Figure 1. The second type of routine provides the behavior of the object. These functions are usually called *methods*. More than simply specifying a property, a method generally causes some sort of action to occur on the object, or requests the object perform some computation. In Figure 1, method routines are prefaced with an (M) before their name (e.g., Delete method of a Page). The ellipsis (…) indicates that more property and method routines are likely present. For simplicity, arguments for the properties and methods have been omitted in Figure 1. Therefore, the state of the object is defined by its properties and the behavior is defined by its methods. Occasionally there is no clear distinction between property and method functions, as a method may return some value that may appear to be a property.

In addition to the state and behavior, objects have a unique identity. There are many ways to establish the identity of an object within an application. One common method is by the address of the object within the memory of the computer. Since no two distinct objects can occupy the same memory, the value of the memory address can differentiate one object instantiation from another object instantiation of the same or different type within the application. Referring to Figure 1, all objects (Drawing, Pages, Page 1, Page 2, …, Shapes, Shape 1, Shape 2, …, etc.) would occupy different memory addresses and consequently have separate identities.

Consider a program object that represents an excerpt of a pump in a process simulator as shown in Figure 2. Here two interfaces for the pump have been defined, IUnitOp and IPump (the leading "I"

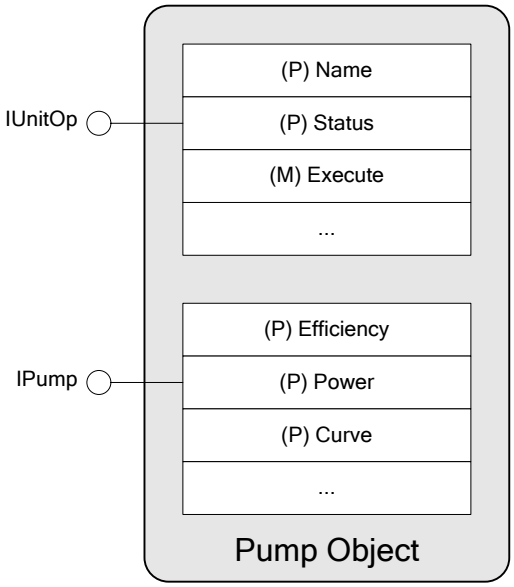

Figure 2. Excerpt of a Pump Object in a Process Simulation Program.

in the name is commonly used to name an interface). The symbol ○—— is commonly used to illustrate the definition of an interface as a connection point for a client application. Again, for simplicity, the arguments for all properties or methods in IUnitOp and IPump in Figure 2 are omitted. The IUnitOp interface is an abstract interface that is common to all unit operations in the simulator. Other unit operations, in addition to the pump, expose the properties and methods defined in the IUnitOp interface, because all unit operations have a Name and Status property, as well as an Execute method. Even though the definition of the IUnitOp interface is the same from one unit operation to another, the behavior may or may not be the same. For example, the Execute method of a pump would behave quite differently than the same method in a distillation column. Abstract definitions provide commonality in content, but not in behavior.

In addition, the pump defines its own specific interface that is not common to other types of objects within the application. The IPump interface allows the client application access to the Curve, Efficiency, and Power properties of the pump. Properties such as flow rate through the pump would not typically be considered properties of the pump, but rather properties of the streams attached to the pump. As was the case for some of the drawing program objects in Figure 1, some of the properties in IUnitOp and IPump are likely to be objects themselves rather than simple values. For example, with pump power, the child object could provide information on units in addition to the numerical value of

the power. The Curve property would quite likely refer to an object that provides data required to define a pump curve such as efficiency, flow rate, and head.

Depending upon object implementation, the IPump interface may be a completely separate interface from the IUnitOp interface, or the IPump interface may *inherit* or *derive* from the IUnitOp interface. If the implementation is made using inheritance, the IPump interface will contain the state, behavior, and identity of the IUnitOp interface as well.

The outer boundary in Figure 2 illustrates the encapsulation of the pump object. Within the pump boundary, the implementation of the state, behavior, and identity are encapsulated, hidden from the client. This encapsulation provides all of the data (variables) and routines (code) required to implement the public state and behavior exposed through the interfaces. In general, the implementation of the pump would almost certainly require the use of other physical and abstract objects within the framework of the application and operating system to provide this encapsulation.

In addition to the interfaces shown in Figure 2, which require the client to make calls to the pump object, a callback or event type of interface may also be defined to allow the object to communicate changes back to the client. The event or callback is typically used to notify the client of changes that have occurred in an object that are not a result of the client directly. For example, as the simulator performs its calculations, the Status property of the pump may change as a result of the progression of calculations. An event could be triggered or fired to notify the client of this change so the client can act accordingly.

Figure 3 extends Figure 2 to show the addition of the client and an event interface. Here, the

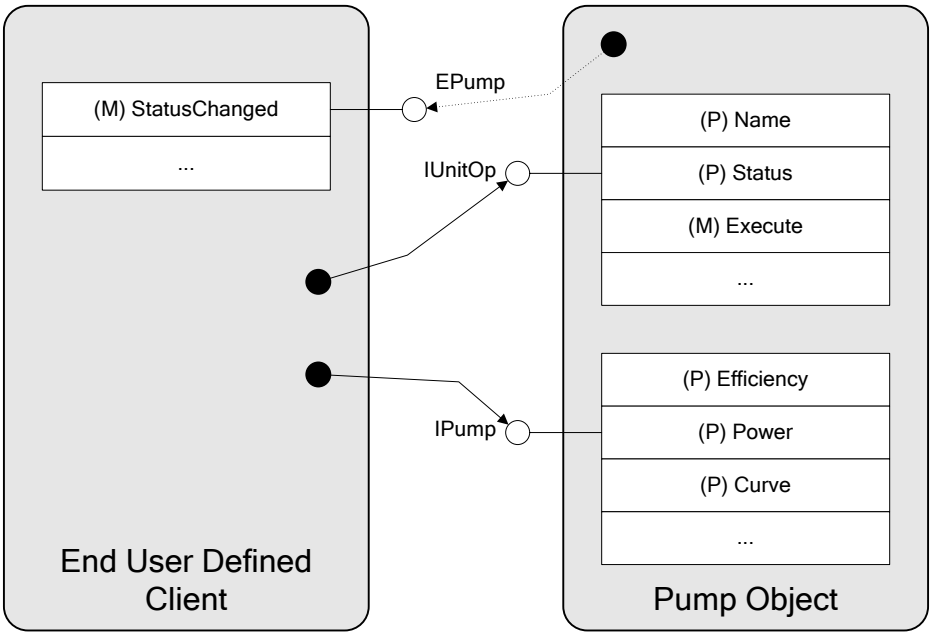

Figure 3. Client Connection in End User Defined Code to Pump Object in Process Simulator with Event Callback.

event interface (EPump—"E" indicating event interface) is implemented by the client of the pump and the pump connects to the client through the interface. The pump can then notify the client of status changes by calling the StatusChanged event method implemented by the client. The StatusChanged event method would likely be given the previous status of the pump as an argument to the event function in case this value is needed by the client. Note that while the content and structure of the

event interface is defined by the author of the pump object, the implementation is actually made in the client application. Depending on the language of the client application, event interfaces may not require implementation if the client does not desire to receive the event.

In addition to the event interface of the pump, the drawing program presented in Figure 1 could fire events whenever an action was made in the application that caused a shape to be changed. For example, if the user moved the shape a corresponding event would likely be fired providing information to a client concerning what shape was moved and possibly its previous location in its function argument list. If the event is fired after the move actually occurs, the new location would probably not be provided in the argument list since that information would be available through a property of the shape.

As mentioned earlier, in addition to objects that have a physical representation, such as the pump, abstract objects are also common in an implementation. As an example, consider the definition of a Font object illustrated in Figure 1. While there is no physical representation of this Font object, the object does provide font state that would be required to implement text within the shapes of a drawing application. Other examples of abstract objects could include thermodynamic and physical property calculator objects within the process simulator that defined the pump in Figures 2 and 3.

AUTOMATION AND COM

Automation (which was originally called OLE Automation) is a Microsoft Windows platform dependent technology that allows an application to expose its internal objects for use by another application, or a macro-programming environment such as Visual Basic for Applications (VBA) or JScript. VBA is a programming language syntactically similar to Visual Basic present in many programs including the Microsoft Office suite (i.e., Microsoft Word, Microsoft Visio®, etc.) of applications. Automation is based on another Microsoft technology, the Component Object Model (COM). Although there are other competing technologies to COM such as CORBA from the Open Software Foundation (OSF), this discussion will focus on COM because of its dominant availability in applications used for engineering. COM is a totally object-oriented protocol that connects two or more software modules together via COM defined interfaces. Usually each module or application has one or more objects, each supporting one or more interfaces. An automation client can only access the services of a COM object by invoking the properties or methods on one of these interfaces. The automation client cannot access the information encapsulated within the implementation of the object.

The IUnknown Interface

COM is founded on the idea that an interface is totally separate from the object implementation. Interfaces are abstract definitions that could allow their implementation to be made in more than a single object (such as IUnitOp in Figure 2); even though in many cases the interface definition is only intended to be implemented by a single object (such as IPump in Figure 2).

Every COM object must support a COM defined interface named IUnknown. The IUnknown interface consists of three methods as shown in Figure 4. The strict structure of the IUnknown interface allows the implementation of a COM client or server to be independent of programming language. The language merely has to support implementing interfaces with functions specified in the order they appear in Figure 4. The object in the server could be written in C++, while the client is in VBA in one of the Microsoft Office products.

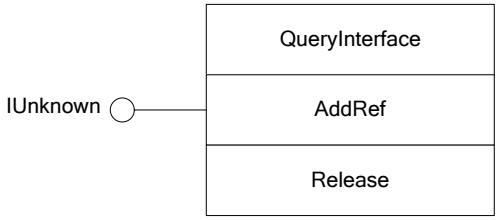

Figure 4. The IUnknown Interface Definition Under COM

Not only must an object support IUnknown, the IUnknown routines must be the first routines in every interface an object exposes. This is usually accomplished by making other interfaces inherit or derive from IUnknown. As indicated earlier, when one interface derives or inherits from another interface, the routines of the base interface are included in the derived interface as the first routines of the interface. Therefore, if the pump object of Figure 2 is implemented as a COM server rather than simply as a generic object, the pump would have to appear as in Figure 5.

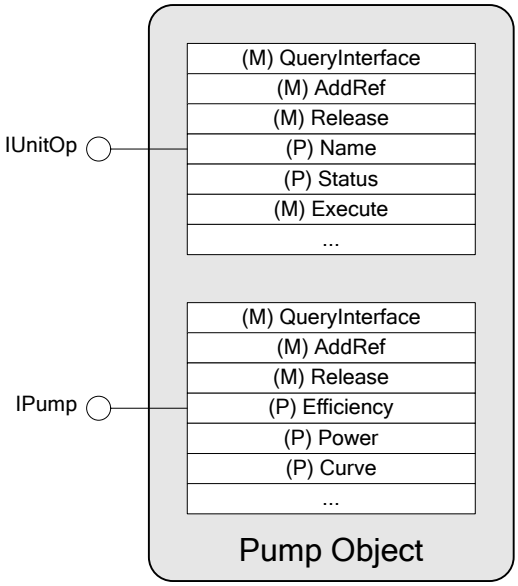

Figure 5. Implementation of the Pump Object under COM

QueryInterface is a simple method, but may be the most important concept in COM. It provides total navigation ability among the interfaces supported by a COM object. QueryInterface is used to request another interface supported by an object. For example, a QueryInterface call on the IUnitOp interface could be made to obtain the IPump interface and vice versa. If an interface is requested that the object does not support, the object returns a COM defined error condition to the client application. For example, if IDistill defines an interface used by a distillation column, requesting IDistill from any of the pump object interfaces would return this error.

The AddRef and Release methods are used for managing the lifetime of a COM object. Each COM object internally must maintain a reference counter that indicates the number of interfaces it has provided to external clients. Consequently, anytime a successful QueryInterface call is made and the requested interface is returned to the client without error, QueryInterface internally calls AddRef to increment the reference count by one. AddRef is also available for clients to call externally as

required. After a client is finished using the interface, the client must call Release to decrement the reference count by one. When the reference count reaches zero, external clients no longer require the object and the object may release resources and remove itself from memory.

COM Client Examples using VBA

Fortunately, when using high-level languages such as VBA in the Microsoft Office applications, the intricate work of managing the IUnknown interface is automatically handled by the programming language environment. In fact, the IUnknown methods are restricted methods so they do not even appear directly in these programming environments. Consider the following code excerpt that sets the first row and column pair (cell A1) of Sheet1 in the Excel Workbook to a value of 5:

```
1)  Sub SetA1()
2)      Dim R As Range
3)      Set R = ThisWorkbook.Worksheets.Item("Sheet1").Range("A1")
4)      R.Value = 5
5)      Set R = Nothing
6)  End Sub
```

The line numbers in the code are present only for the discussion here. They must be removed before actually using the code. The period (".") operator in line 3 is used to call properties and methods for an object. The name to the left of the period represents the object and the name to the right represents the property or method to call. Recall that properties (and sometimes methods) can return objects themselves. (Most Automation objects only expose a single interface to the client application, and this interface is typically named based on the object itself. Unfortunately, this one-to-one correspondence may lead to ambiguity between an interface and an object within the discussion of client implementation. Therefore, when a property or method indicates an object is being returned, it actually means that an interface to an object is being returned and this interface is the only exposed interface for the object.) This fact allows the syntax in line 3 to traverse the child objects of the ThisWorkbook object because at each level through this statement a child object is returned that can in turn have its properties or methods called. Each line of the sample is discussed in detail below.

Line Number	Purpose
1	Declares the start of a macro called SetA1.
2	Declares a variable R that represents a Range interface.
3	A. Call the Worksheets property on the ThisWorkbook object to obtain its Worksheets collection. The ThisWorkbook variable is automatically defined by Microsoft Excel to represent the Workbook object. B. Using the Worksheets collection from step A, call the Item method passing a Sheet1 argument to obtain the Worksheet object representing Sheet1 in the Excel Workbook. C. Using the Worksheet object for Sheet1 in step B, call the Range property for the Worksheet and return an object that represents the range for cell A1. D. Call QueryInterface to obtain the Range interface and assign it to variable R. The Set statement in VBA internally calls the IUnknown method QueryInterface.
4	Set the value of Sheet1!A1 to 5 using the Value property of the Range interface.
5	Calls the Release method of IUnknown to decrement the reference count on the object obtained by the QueryInterface call in line 3 part D.

Line Number	Purpose
6	Marks the end of the macro.

To add this code to Microsoft Excel, start the VBA editor environment by selecting Tools->Macros->Visual Basic Editor from the Excel menus. From the VBA environment, select Insert->Module to insert a code module then add the code to the newly created module. You can then run this code by switching back to the Excel workbook view and select the Tools->Macros->Macros… menu and run the SetA1 macro by selecting it in the dialog that appears.

In order to assist the user in developing macros, a special tool known as the Object Browser is available to provide the object hierarchy in an application. From the VBA environment, the Object Browser is available through the View menu. Using this tool, the user can discover that the Worksheets collection is a property of a Workbook object, that Item is a property of the Worksheets collection, and that Range is a property of the Worksheet object.

In addition to the Object Browser, VBA offers a feature known as IntelliSense to assist the user further in determining the properties and methods available on an object. As the code is being typed into the VBA editor, IntelliSense will display a list box of the possible property and method names when the period is typed in the statement. IntelliSense also indicates the names of the arguments that are required when making a property or method call as the code is being typed. IntelliSense technology appears at other key locations when writing VBA code to provide assistance. IntelliSense and the Object Browser provide helpful information provided a registered type library is present for the object and the object returns interfaces defined in the type library. Unfortunately, there is a COM abstract interface named IDispatch that is occasionally returned by an object. When IDispatch is returned, neither the Object Browser nor IntelliSense can provide the user assistance as to the availability of properties and methods. Only the product documentation can help here. A full discussion of this scenario is beyond the scope of this presentation. The reader is encouraged to refer to product documentation for more information on IDispatch.

As a more practical example, consider the code present in Listing 1 which provides an example of collaboration between Microsoft Excel and BR&E ProMax®, Bryan Research & Engineering's general purpose process simulation program. This example illustrates the generation of complete saturated steam table data (saturation temperature and pressure, and molar volume, molar enthalpy, and molar entropy for the saturated vapor and liquid) from the water triple point to the critical point using the NBS Steam Tables package in ProMax. This code is run using the VBA environment in Microsoft Excel, as was the simple example above. The example illustrates that with a small amount of code, the user can exploit the power of Excel and ProMax through their exposed objects to provide quite extensive data. After running this code, the data would be available for charting or any other desired purpose in Sheet1 of the Excel workbook.

While a complete description of the implementation details is beyond the scope of this presentation, the reader should have an idea of what can be accomplished using Automation and software available to them. There are many books that deal with writing macros and other utilities in Microsoft Office applications that should serve as reference for those desiring to exploit this technology.

Monikers

In this paper, we have discussed and presented examples of the parent-child-sibling relationship that is present between objects in an application. An additional way of uniquely identifying an object is through a textual name that specifies the object through its ancestors. In COM, this name is called a

```
' Generates saturated steam table data in SI units using NBS Steam Tables property package in
' BR&E ProMax and stores the results in Excel Worksheet #1.  Requires a VBA project reference
' to the BR&E ProMax type library in the Tools->References menu of the VBA environment.
Sub NBSSteam()
    Dim PMX As New ProMax.ProMax
    Dim Env As ProMax.Environment

    Set Env = PMX.New.Environments.Add
    Env.PhysPropMethodSet.LoadPackage "NBS Steam Tables"
    Env.Components.Add "Water"
    Dim PP As ProMax.PhysProp
    Set PP = Env.PhysProp

    Dim T As Double, P As Double, Status As Long, R As Integer
    R = 1                                   ' Start in Row 1 of Sheet1 of ThisWorkbook
    For T = 273.16 To 647.126               ' Triple point to critical point (K)

        ' Get vapor pressure at current temperature T
        P = PP.CalcPure(pmxVaporPressure, pmxLLiquidPhase, pmxUseTemperature, Status, T, P, 0)
        Sheet1.Cells(R, 1).Value = T            ' Temperature (K) in column 1
        Sheet1.Cells(R, 2).Value = P            ' Pressure (Pa) in column 2

        ' Calculate saturated liquid molar V, H, S in columns 3, 4, 5
        Sheet1.Cells(R, 3).Value = _
               PP.CalcPure(pmxMolarVolume, pmxLLiquidPhase, pmxNullPropMask, Status, T, P, 0)
        Sheet1.Cells(R, 4).Value = _
               PP.CalcPure(pmxMolarEnthalpy, pmxLLiquidPhase, pmxNullPropMask, Status, T, P, 0)
        Sheet1.Cells(R, 5).Value = _
               PP.CalcPure(pmxMolarEntropy, pmxLLiquidPhase, pmxNullPropMask, Status, T, P, 0)

        ' Calculate saturated vapor molar V, H, S in columns 6, 7, 8
        Sheet1.Cells(R, 6).Value = _
               PP.CalcPure(pmxMolarVolume, pmxVaporPhase, pmxNullPropMask, Status, T, P, 0)
        Sheet1.Cells(R, 7).Value = _
               PP.CalcPure(pmxMolarEnthalpy, pmxVaporPhase, pmxNullPropMask, Status, T, P, 0)
        Sheet1.Cells(R, 8).Value = _
               PP.CalcPure(pmxMolarEntropy, pmxVaporPhase, pmxNullPropMask, Status, T, P, 0)

        R = R + 1                           ' Move to next row for next temperature
    Next T

    Set PP = Nothing
    Set Env = Nothing
    Set PMX = Nothing
End Sub
```

Listing 1. Sample VBA Code in Microsoft Excel for Creating Saturated Steam Data from NBS Steam Tables in BR&E ProMax. Illustrates collaborative use of BR&E ProMax object model and Microsoft Excel object model.

Moniker. There are various types of monikers available in COM, but this paper will only introduce the composite moniker.

Monikers can be considered analogous to a file name path in the file system. Consider a file and directory path name such as:

C:\My Files\My Subfolder\My File.txt

In this file name, the backslash character ("\") is used to separate the entities in the ancestry of the file "My File.txt". The top item in the ancestry is the C: representing the C disk drive. The next component represents the "My Files" subfolder present as a child of the C: drive. "My Subfolder" represents a child folder of "My Files", and "My File.txt", a child of "My Subfolder", represents the desired file. The complete path uniquely identifies the file object in the file system among all of the

other files and folders present. With a composite moniker, an application defines the separating character and the object is named using its ancestry. In many applications, the character used as a separator is the exclamation point (!). By using the name of the moniker in functions that accept monikers, a client can access the objects of an application. In many cases, the access can even be across a process boundary because the moniker is a unique name for the object within the entire system, not just the application.

Consider the following code excerpt from VBA in Microsoft Excel to display the power of a glycol pump in a running BR&E ProMax simulation.

```
Sub DisplayPower()
    MsgBox GetObject("ProMax@C:\ProMax\Examples\Dehydration\Basic Dehydration Unit.pmx!" & _
                     "Flowsheets!Flowsheet1!Blocks!Glycol Pump!Properties!Power")
End Sub
```

The VBA intrinsic function GetObject will return the object represented by moniker name supplied as an argument. The moniker name, though long, is relatively simple to comprehend in content. (The ampersand operator in VBA concatenates two strings into a single string. The underscore at the end of the line allows the line of code to be broken in the middle for better display.) The name uniquely identifies the pump that is desired within the context of all applications running on the system. The initial part of the moniker identifies the particular case or file within the context of all running instances of ProMax, and the remainder of the moniker identifies the object within that specific case. The syntax above is only included for illustrative purposes. The syntax of the moniker string is application dependent. The reader will need to consult product documentation or the application vendor for the syntax of a particular program. It should be pointed out that moniker access to the objects within an application is not available in all programs. This capability must be designed and implemented in the application. The reader is encouraged to refer elsewhere [4] for more information on monikers in COM.

EXTENSIBLE MARKUP LANGUAGE

The World Wide Web Consortium (W3C) first standardized Extensible Markup Language (XML) in February 1998, to help combat the complexity of large-scale electronic publishing. XML's origins, however, date well beyond this to the 1980s with the development of Standard Generalized Markup Language (SGML, ISO 8879:1986). It was at SGML 96 Conference in Boston (November 1996) that XML first took shape in the form of a draft specification. The designers envisioned a text based language used purely to structure, store and to send information in a vendor independent format.

The goals laid out in the SGML 96 Conference are as follows:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.

9. XML documents shall be easy to create.
10. Terseness is of minimal importance.

XML's simplicity and reasonable legibility, combined with platform-independence, has given it popularity well beyond the majority of its peers. Not only has it outgrown electronic publishing, but is also used for a wide range of applications beyond simple data storage and transportation. Protocols such as Simple Object Access Protocol (SOAP) use XML to invoke methods on remote servers while a variety of applications use XML during deployment and for configuration. Its use extends to represent relational databases (viewable without programming) in web browsers, creating document templates and a variety of other possibilities that reach almost every corner of the computing world. Because of its rapid acceptance, this simple language, with its extensible grammar, is now at work in a growing list of companies and applications, and is virtually certain to be used on your personal computer in some fashion.

The reader is cautioned that many of the XML supporting technologies are still emerging and any cited references may not be fully current with these standards. Current specifications are available on the W3C web site [5].

Basic XML Structure

The XML standard defines the basic structure of an XML document, but not the content. The content is determined by the publisher or consumer of the document. To explore how XML may be used, consider the simple XML document in Listing 2 that stores chemical compound data. Notice how this is analogous to a database that stores properties for chemical compounds.

Observe the use of angled brackets ("<>") surrounding names throughout this example. These tags, as they are frequently referred to in markup languages, demarcate the various parts of the document. Since a variety of markup languages have this trait, it is necessary to indicate to the document reader that the format of the file is XML 1.0. This is the purpose of the *document prolog*, represented by the first lines of any XML file and contains tags that start and end with "`<?`" and "`?>`", respectively.

Immediately after the document prolog is the *root* element called `MyChemicalData` in Listing 2. There can only be one root element and it encompasses all other data elements. Elements begin with an opening angle bracket ("`<`") followed by a user selected name. An element name cannot contain any spaces or special characters. Also, note that XML is case-sensitive. Termination of an element containing data or child elements is indicated by a forward slash ("`/`") preceding the element name in angle brackets. When no data or child elements are present, the forward slash and closing angle bracket ("`/>`") may be used to terminate the definition of the element. In XML, every element definition must be terminated. Therefore, the definition of the root element continues until the last line of the document in Listing 2, where the termination sequence is encountered.

Relationships within the data are represented by the notion of whom I reside in is my parent and I am a child of that element. Therefore, the `MyChemicalData` element is the parent of all `Compound` elements and `Compound` elements have `PhysicalProperty` elements as children. Following this idea further, siblings all reside at the same level as each other within a single parent. This last restriction prevents any element overlapping with any other element, thus preserving the parent–child relationship throughout the entire structure, again a common trait among object-oriented methodologies. In Listing 2, `Compound` and `PhysicalProperty` elements all have sibling elements.

An element definition may also contain any number of uniquely named entities called *attributes*. Examples of attributes in Listing 2 include the `version` attribute in the prolog line and the `Name` and `Formula` attributes in each `Compound` element. Just like element names, attribute names

```
<!-- Example XML file for Compound Physical Properties -->
<?xml version="1.0"?>
<MyChemicalData Creator="Me" Version="1.0" Updated="01-01-2005">
   <Compound Name="Methane" Formula="CH4">
      <Alias>Methyl hydride</Alias>
      <Alias>Marsh gas</Alias>
      <PhysicalProperty Name="Triple Point Temperature">
         <Value Units="K" Uncertainty="0.01" Reference="Kleinrahm, R.;
          Wagner, W., Measurement and correlation of the equilibrium liquid
          and vapour densities & the vapour pres. along the coexistence
          curve of methane, J. Chem. Thermodyn., 1986, 18, 739-
          60.">90.68</Value>
      </PhysicalProperty>
      <PhysicalProperty Name="Triple Point Pressure">
         <Value Units="bar" Uncertainty="0.0001" Reference="Younglove, B.A.;
          Ely, J.F., Thermophysical Properties of Fluids II. Methane,
          Ethane, Propane, Isobutane, and Normal Butane, J. Phys. Chem. Ref.
          Data, 1987, 16, 577.">0.1174</Value>
      </PhysicalProperty>
         ⋮
   </Compound>
   <Compound Name="Ethane" Formula="C2H6">
      <Alias>Ethyl hydride</Alias>
         ⋮
   </Compound>
      ⋮
</MyChemicalData>
```

Listing 2. Example Illustrating Basic XML Structure Using Excerpt from a Chemical Compound Database. Note: "⋮" represents additional lines removed for simplicity. Reference attributes are split across multiple lines in this display. In reality, there is no split in the original file.

cannot contain spaces or special characters. An equals sign ("=") separates attribute names and their values, which must be enclosed in either single or double quotes. Special characters in attribute values such as ampersand, angle brackets, and quotes must be specially treated.

Data present between beginning and ending definitions of an element, not enclosed in angle brackets, is called the text of that element. In Listing 2, for example, `Alias` sibling elements for methane provide alternative names for that compound. A common design dilemma with XML is determining when to create attributes as opposed to when to create child elements and place the attribute text as text within the child element. Many times the decision is somewhat arbitrary, but a general guideline is if it appears like data, use a child element.

As with all programming languages, the addition of comments in Listing 2 improves human readability. In XML a comment starts with "`<!--`" and ends with "`-->`" and can appear anywhere within the document. The first line in Listing 2 provides an example of an XML comment.

XML provides a concept called *namespaces* that are applied to elements or attributes to allow a document to be divided into application specific groups. The namespace allows an application that utilizes the XML data to process only the groups of data that are significant to it as designated by a specific namespace. Additionally, different groups of data may utilize the same name for elements or attributes resulting in ambiguity. The namespace can be used to resolve this ambiguity.

Namespaces are definable throughout the document, but are only valid for use in descendants of the element containing the definition. To use a namespace, each element or attribute name is prefixed with a user-defined namespace prefix followed by a colon (“:”). This prefix is actually an alias for a Uniform Resource Locator (URL), internally used as a unique namespace string identifier, typically of the organization that maintains that namespace. XML keyword `xmlns:` prefixed to an element attribute, along with a URL assignment, denotes a namespace prefix as seen below. Note that

```
<MyPrefix:NodeName xmlns:MyPrefix="http://.../MyURL/"…>
      -content-
</MyPrefix:NodeName>
```

if no prefix name is given, the namespace becomes the default for all descendants.

For an example of namespace implementation, consider Listing 3. This XML document, originally created by Microsoft Excel 2003, highlights various XML features already discussed. Notice there is a second line in the document prolog to tell the reader that the contained data are for a Microsoft Office (mso) Excel Worksheet. The root element, designated `Workbook`, contains three namespace definitions, one default, one aliased as `x`, and another as `ss`. These named aliases prefix several element attributes in several places where confusion could occur. A good example is the attributes of the `Table` element.

Focusing on the overall structure, it becomes apparent that Excel organizes its XML data in terms of worksheets, tables, rows, and cells. In this example, cell data have a designated type and possibly a formula. When loaded, this workbook should contain a worksheet called `Sheet1` containing three cells with data. The first contains the number “1”, the second “2”, and the third is the sum of the previous cells. As a side note, if these cells were not adjacent, the attribute `ss:Index="#number"` would be added to `Row` and `Cell` elements to designate the actual row and column (cell) number.

```
<?xml version="1.0"?>
<?mso-application progid="Excel.Sheet"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
 xmlns:x="urn:schemas-microsoft-com:office:excel"
 xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet">
   <Worksheet ss:Name="Sheet1">
      <Table ss:ExpandedColumnCount="1" ss:ExpandedRowCount="3"
       x:FullColumns="1" x:FullRows="1">
         <Row>
            <Cell><Data ss:Type="Number">1</Data></Cell>
         </Row>
         <Row>
            <Cell><Data ss:Type="Number">2</Data></Cell>
         </Row>
         <Row>
            <Cell ss:Formula="=SUM(R[-2]C:R[-1]C)"><Data
             ss:Type="Number">3</Data></Cell>
         </Row>
      </Table>
   </Worksheet>
</Workbook>
```

Listing 3. XML Document for Microsoft Excel.

Validation of XML Data

Since an XML document can originate from anywhere, an application must have some way of verifying the content and structure of XML data it expects. XML is extremely flexible but applications utilizing it usually require rigid data presentations for consumption. For example, an application that is expecting to parse telephone numbers would expect the data within the document to be formatted in a specific manner. This leads to the next important part of an XML document, Document Type Declarations (DTD) and Schemas, the W3C successor to DTD. DTD's and Schemas add definitions of order and data type ensuring consistency in how applications interpret and modify an XML document.

Predating XML, DTD is a legacy of SGML and consist of a set of rules or declarations either within an XML document or as a separate document file. XML validating parsers compare documents to the DTD and list regions where the document differs from the DTD rules. The parsing program can then determine how to handle these conditions. More information on DTD's can be found in Harold and Means [6].

XML Schemas, unlike DTD, are XML documents in their own right and are a type of template. For Schema and target XML documents to work together, both documents must contain certain elements. The XML document root element must first declare a prefix associated with the `XMLSchema-instance` namespace and then use the `schemaLocation` attribute of that namespace to define the target namespace and the schema file location as shown below. Schemas are one of the

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi.schemaLocation="http://www.mysite.com MySchemaFile.xsd" … >
      ⋮
</root>
```

emerging areas of XML and the best reference for them is the W3C documentation on the W3C web site.

Cascading Style Sheets and Beyond

Discussion thus far has centered on creation and storage of data, but that is only half the story. The real beauty of XML comes into play in data manipulation. In the last few years, several languages and tools have been developed to take full advantage of this self-describing data format. In a broad sense, these divide into transformation tools and programming tool kits, with the latter residing almost exclusively in the domain of application programming. Most of these tools are available freely as Internet downloads. For simplicity, only a brief overview follows for a few of the more widely adopted tools.

Consider first the oldest of all the XML transformation tools, Cascading Style Sheets (CSS), initially developed in 1994 for HTML. This technique uses a set of rules to apply word-processing like formatting to XML elements to render a new document. All popular web browsers of today support, to varying degrees, CSS thus adding to the appeal of this technique. Unfortunately, its simplicity limits its usefulness due to its inability to rearrange and massage XML data.

The next and more powerful example of XML transform tools is called Extensible Stylesheet Language (XSL). XSL actually consists of three parts: XSL Transformations (XSLT) [7], XML Path Language (XPath), and XSL Formatting Objects (XSL-FO) [8]. XSLT follows the structure and syntax of XML and provides the ability to transform an XML file into any other type of desired file. XSLT transforms the XML file based on definitions made in an XSL stylesheet, a file that contains the XSLT program code. By programming a stylesheet, XML can be transformed from its original presentation into another XML file or any other text based file for use in a database, for input into Microsoft Excel, for display on a web page using HTML, for display as plain text sentences, etc.

Obviously, each transformation would require a different stylesheet to produce the desired document. With a properly written XSLT stylesheet, the XML code in Listing 2 could be transformed into a Microsoft Word document. For this particular example, this may not seem like much power but consider what this would provide if the XML file contains hundreds of compounds.

XSLT uses XPath as a selection language to select or filter elements for processing and inclusion in the transformation of an XML file. For example, the following simple XPath statement would select the triple point pressure element in Listing 2 for processing in XSLT:

```
//MyChemicalData/Component[@Name="Methane"]/PhysicalProperty[@Name="Triple Point Pressure"]
```

XPath offers significant flexibility in syntax including the use of wildcard characters to provide a powerful selection language. Using the above XPath statement in an XSLT transformation would allow inclusion of only the triple point pressure data for methane in the transformed document. With more complicated XPath statements, the selection could be limited to all components which had triple point pressures below a specified value.

XSL-FO is an XML vocabulary for specifying formatting semantics used to obtain printed output. An XSL-FO file is an XML file that not only includes the data, but also how the data will be represented on paper. XSL-FO is the process for creating a presentation out of a description. The definition allows for quality presentation of data, comparable to what is available in a word processing program. XSL-FO allows for the creation of formatted tables and other complicated output. XSL-FO files are usually created by applying XSLT transformations to a source XML file. The transformation process not only transforms the data for presentation, but also selects the desired data to be included in the presentation using XPath. Microsoft has recently introduced an XSLT stylesheet to transform Microsoft Word 2003 XML files into XSL-FO files.

XSL-FO files are not printed directly. They are still XML files. However, they contain all of the instructions on how to render the presentation. In order to print an XSL-FO file, an additional tool is required. Many of these tools are available freely on the Internet. The tools typically convert the file from XSL-FO into Adobe® Portable Document Format (PDF), Microsoft Rich Text Format (RTF), and others.

Two competing technologies are available for programmatic access to data held in XML files. The most prominent is a programming interface specification called Document Object Model (DOM), developed once again by the World Wide Web Consortium. DOM is an object-oriented approach to parsing and manipulating the data within an XML file. DOM objects represent the entities within an XML file (elements, attributes, etc.), and expose interfaces for a client application to manipulate or read the data. DOM objects contain methods to load and save XML files. DOM will load the entire XML file into memory ensuring the basic structure of XML is adhered to in the document. As with XSLT, through the power of the XPath language, DOM allows flexible and powerful programmatic selection of elements within the XML file. DOM is available on a wide variety of platforms and can be used with VBA clients such as Microsoft Excel (and others) on Microsoft Windows systems.

The second technology for programmatic access is called Simple API for XML (SAX). SAX was originally developed to run as part of Java but is today used in many object-oriented programming languages. SAX is an event-driven, serial-access mechanism for accessing XML documents using the object event firing approach presented earlier in this paper. As SAX processes an XML document, the SAX objects fire events to a client application and the client can process or discard the data. Therefore, SAX, unlike DOM, does not require the entire XML document to be loaded into memory at one time.

SUMMARY

This paper has introduced many of the software technologies used today in a variety of applications used by engineers. These technologies include Object-Oriented Programming, OLE Automation, and XML. Object-oriented programming is the fundamental basis of all these technologies. Automation provides a tool to extend or manipulate an application beyond its original design. XML is a data presentation technology that offers flexible and powerful validation, transformation and presentation capabilities, along with object-oriented programming models to access the data in client applications. An attempt has been made to provide discussion and examples that illustrate these technologies so that the reader can determine if a specific technology is applicable to a task. No attempt has been made to provide a full description of the implementation of these technologies with the context of this paper. Rather, the reader is encouraged to consult the references and product documentation for implementation details.

Finally, we should note that a newer object-oriented programming technology for the Microsoft Windows platform will likely be prominent in the near future, namely Microsoft .Net. However, at this time, few large-scale applications (including Microsoft Office) utilize .Net. Consequently, .Net was not a focus for this paper.

REFERENCES CITED

1. Stroustrup, B., *The C++ Programming Language, Third Edition*, Addison-Wesley, Reading, MA, 1997.

2. Smith, M., and Tockey, S., "An Integrated Approach to Software Requirements Definition Using Objects," Boeing Commercial Airplane Support Division, Seattle, WA, 1988.

3. Booch, G., *Object-Oriented Analysis and Design with Applications, Second Edition*, The Benjamin/Cummings Publishing Company, Inc, New York, 1994.

4. Brockschmidt, K., *Inside OLE, Second Edition*, Microsoft Press, Redmond, WA, 1995.

5. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., "Extensible Markup Language (XML) 1.0 (Third Edition)," W3C Recommendation, Feb. 04, 2004.

6. Harold, E.R., and Means, W.S., *XML in a Nutshell—A Desktop Quick Reference*, O'Reilly & Associates, Sebastopol, CA, 2001.

7. Kay, M., *XSLT, 2nd Edition*, Wrox Press, Birmingham, U.K., 2001.

8. Pawson, D., *Making XML Look Good in Print: XSL-FO*, O'Reilly & Associates, Sebastopol, CA, 2002.