The SmartPlant Adapter

Authoring Course Guide

August 2007 Version 3.8

DSPF1-TP-100003C

The SmartPlant Adapter

Authoring Course Guide

August 2007 Version 3.8

DSPF1-TP-100003C

Intergraph Corporation
Huntsville, Alabama 35894-0001

This courseware was developed by Bill Crego and Dell Wilson, PPM-PIM Training, Huntsville, Alabama.

Table of Contents

S E C T I O N 1

Schema Component Programming

1. **VB Programming with Interfaces**
2. **SmartPlant Architecture Overview**
3. **Containers and Container Compositions**
4. **Retrieving Objects in Containers**
5. **Traversing Relationships**
6. **Intrinsic versus Application Interfaces**
7. **Creating New Objects and Relationships**

1. VB Programming with Interfaces

1.1. What is an interface?

Interfaces are part of Microsoft's *Component Object Model* architecture, known as COM. COM provides a foundation allowing all different types of software and applications to interact and share data and functionality. Think of COM as a set of rules and concepts; as long as a programmer follows COM rules, reusing and sharing functionality via component software (DLLs, OCXs, etc.) can become a reality.

Two of the bedrock technologies employed by COM are *interfaces* and *type libraries* (typelibs).

Interface	An interface is a list of closely related functions implemented by an object. It constitutes a contract between an object and any client using the object.
Type Library	A type library is a standardized description of the interfaces supported by an object. This includes the names of the methods, their parameters, parameter types, returns, etc.

How Visual Basic uses type libraries and interfaces

Most Visual Basic .NET programmers have referenced a component via the *Project->References* pull down menu in the Visual Basic .NET GUI. Usually, the programmer selects or browses for a DLL containing a certain component (or class) containing some desired functionality. A common example is the *Microsoft Scripting Runtime* library. This DLL contains a class called the *FileSystemObject*, which makes it easy to manipulate files on disk, perform I/O, etc.

What the user is actually accessing inside the DLL is a typelib. In this case, the programmer is creating software (acting as a client) and is using the services of the referenced object.

The service component supports an interface called *IDispatch*. In Visual Basic .NET, any referenced object exposes its functionality (methods and properties) via the IDispatch interface. Visual Basic .NET creates this interface automatically when a VB project is compiled (along with a typelib embedded inside the DLL). For C++ DLLs, there is a language/file extension called IDL that describes the classes and interfaces supported by an object, including the functions exposed via IDispatch.

All Visual Basic .NET programmers have used typelibs and interfaces, even if they have been unaware of it!

1.2. Providing a service (vs. consuming)

What if a programmer is required to write a component that serves as an object (service) to be consumed by other clients? What if the client piece of software defines the functionality required from the service object? This is one place where interfaces and the *Implements* keyword come into play in the Visual Basic .NET environment, and this is exactly the scenario presented when authoring a SmartPlant Adapter Component. In order to integrate an application into the SmartPlant Enterprise, the programmer is required to create certain service objects to fulfill certain key roles required by SPF and the SmartPlant.

The SmartPlant creates/maintains specifications for the service object(s) required for this application integration. These specification(s) are in the form of a typelib. The typelib specifies all of the interfaces that the service object may implement. The service object does not necessarily implement all of the interfaces, but it does have to construct every function contained in an implemented interface, even if that function is just a stub (empty).

IDL (along with the mktyplib utility) or the VB language may be used to create typelib files. These typelibs have no code behind them, no functionality. They simply provide the specification for the functionality required by the SmartPlant. For more information on the mechanics of creating typelibs and defining interfaces, see the following topics in MSDN:

Creating Standard Interfaces with Visual Basic
Implementing and Using Standard Interfaces
MkTypLib

The Visual Basic .NET programmer implements interface(s) inside a Class module only. Once the programmer uses the *Implements* keyword, he or she is agreeing to implement all of the functionality specified in the interface. Interfaces in the referenced SmartPlant typelibs are identified by the capital letter "I" that precedes the name in the *Object Browser Classes* list (e.g., *IContainer*). Once the programmer uses the following syntax:

```
Implements IInterfaceName
```

The interface name is added to and becomes selectable in the object (left) dropdown list at the top of the VB Code Editor window. Once selected, all of the functions in the interface are listed in the procedures (right) dropdown list. Each of the functions *must be implemented* in the class, even if the function(s) remain empty.

1.3. Why Interfaces?

There are multiple advantages to using interfaces within the COM architecture. One is they provide a way to organize closely-related functions into one discrete group. They enable polymorphism without requiring inheritance, which can get complicated and confusing the deeper the inheritance scheme runs.

They also provide a means for incremental or evolutionary development, plus there is no need to recompile every component in the system when an interface changes. Compatibility can be maintained because new versions of a component continue to provide existing functionality via existing interfaces, while adding new or enhanced interfaces doesn't break existing clients.

Because interfaces are implemented inside classes in the VB environment, you can Dim interfaces just like Class objects, New them, etc. On the client (consumer) side, an interface is treated just like a class, so there is nothing new for the VB programmer to learn.

The biggest advantage is simply stating and following the contract that an interface implies. All of the pieces of software will work properly if all of the rules are followed, and interfaces force proper syntax and callability.

1.4. Lab Exercise

Review the Chapter 2 – Interfaces Visual Basic .NET program (located in *D:\SMARTPLANTAdapterClass\SchemaComponentProgramming\Chapter 2 – Interfaces*) that demonstrates the use of interfaces. The example covers:

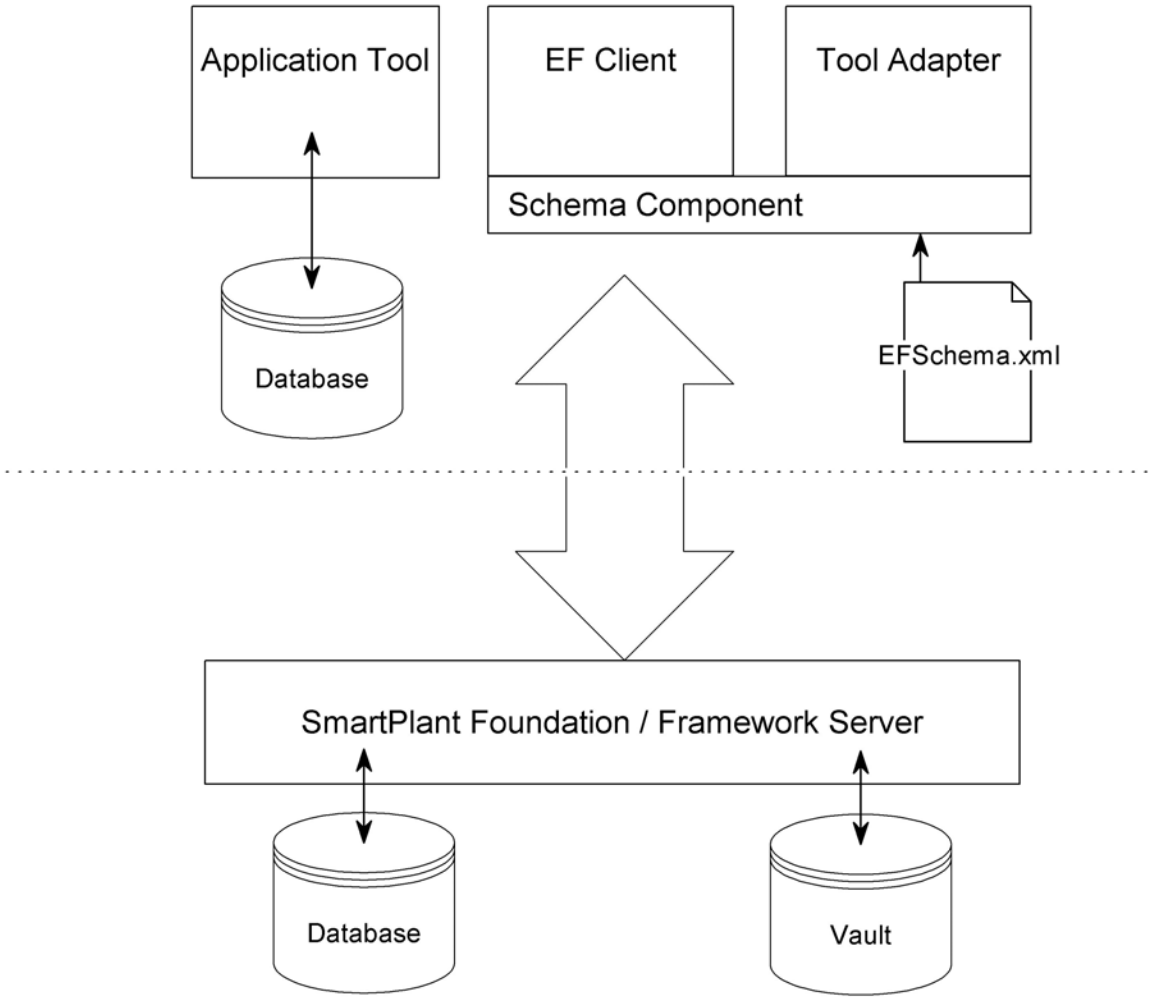
- Creating an abstract interface definition
- Providing a service by implementing the interface
- Consuming the service object created

2. SmartPlant Architecture Overview

The SmartPlant Enterprise is a set of software built on top of SmartPlant Foundation (SPF). SmartPlant enables the sharing of data between applications (using SPF as the central data repository). An application must meet the following requirements to participate within SmartPlant:

- Model its data within the SmartPlant Schema (EFSchema.xml file, using the Schema Editor).
- Implement commands inside the Application Tool to call SmartPlant's Register, Publish, and Retrieve UI. This sets up and enables data transfer between the application and SmartPlant.
- Develop a Tool Adapter to support communication within SmartPlant.

The following graphic provides an overview of the components involved:



Intergraph provides the following software for enabling an application to interact with SmartPlant Enterprise:

- SmartPlant Client
- Schema Component
- SmartPlant Foundation (SPF)
- SmartPlant Server and Loader

The Application must provide the following:

- The Application Tool
- Application schema file (.xml)
- The Tool Adapter

2.1. TEF Components

The Engineering Framework is comprised of the following components:

- **SmartPlant Client —**An ActiveX .dll that allows an application to register with EF, connect to EF, and publish and retrieve data. The SmartPlant Client has three main responsibilities:

  1. Provides common user interface (UI) components to enable an application to register, publish, and retrieve in a consistent fashion.
  2. Calls interface methods on the Tool Adapter during publish and retrieve.

     ***Note**: it is the SmartPlant Client that communicates with the Tool Adapter, not the application tool itself.*

  3. Handles passing data between the Tool Adapter and the SmartPlant Server.

- **SmartPlant Server** — The SmartPlant Server takes requests from the SmartPlant Client and communicates with SPF. It returns the results of these requests back to the SmartPlant Client. This flow of data enables publishing, retrieving, and comparing data. The SmartPlant Server is installed on the SPF server.
- **Framework Loader**—Loads data published by the application(s) into SmartPlant Foundation.
- **Schema Component (SC)** —A suite of ActiveX components that provide functionality surrounding the creation, parsing, validation, and comparison of the Framework schema and data. Defines interfaces used while developing the Tool Adapter

2.2. The SmartPlant Schema

The EFSchema.xml file describes the structure of data passed through SmartPlant along with its rules. The EF schema can be hard to understand; to make it easier to interpret, the **Schema Component** exists.

The Schema Component is a set of .DLLs that assists the tools with the generation and subsequent parsing of the XML data. The tool adapter interfaces with the Schema Component (the main interface point) to read the Framework schema. The following rules for the EF Schema must be followed:

- Application data must be modeled within the TEF Schema.
- The EFSchema.xml file must exist on the server and every client.
- All data processing supported by TEF must be modeled in the TEF Schema file.
- A component schema file is a subset of the TEF Schema that represents the data modeling for one application.

2.3. Identity in TEF

Applications (and their objects) must maintain a unique ID or signature for the duration of their lifetimes within the SPF/SmartPlant Enterprise. There are certain rules governing uniqueness within SPF:

1. Every object must be unique (UID). The scope is within the SPF installation, not globally.
2. An object's UID must never change.
3. Once an object is deleted, its UID may not be reused (no UID or object pooling).

Applications receive a unique *signature* back from SPF during registration. This signature may be combined with an application's own UID system to ensure uniqueness within SPF. For example, signatures begin with strings like "AAAA", "AAAB", and so on. Combine signatures and application UIDs results in a string like "AAAA_appl_id".

3. Containers & Container Compositions

3.1. Containers

A *Container* is an object used to pass data back and forth between an application and SPF. A container may hold data or metadata (data about other data) related to the data model or actual instance (tag or asset) data. A container can hold information loaded from an .xml file or from the SPF database. In the context of creating a SmartPlant Adapter, the containers usually are provided from SmartPlant. During a *Retrieve* operation, the adapter reads data from containers and then takes appropriate action. During a *Publish* operation, the adapter receives empty containers from SmartPlant and fills them up with data to be published.

Note: *The exercises in this chapter demonstrate manipulating data loaded from an .xml file into a container. Though not common in developing a SmartPlant Adapter, this technique is sufficient for showing how to manipulate containers and their contents.*

A container has a particular scope that describes the type of information it contains. That scope can be determined by the *scope* property of the container. There are a number of container scopes defined in SmartPlant; however, we will discuss only the following three: *MetaSchema*, *Schema*, and *Data*.

MetaSchema Containers

MetaSchema containers contain information that describes the types of schema objects and data types available. MetaSchema is the topmost level in the hierarchy of container scopes.

MetaSchema scopes schema object types like Class Definitions (*ClassDefs*), Interface Definitions (*InterfaceDefs*), Relation Definitions (*RelDefs*), and all of the other object types you learned about in the *SPF Modeling and Schema Editor* course. Though you can extend the metaschema by defining your own object definition types, we will use the metaschema "as is" in this class.

Schema Containers

Schema containers hold the schema objects defining the data model (i.e., objects defined within either the EFSchema.xml or your component schema .xml file). In most cases when developing your application's SmartPlant Adapter, you will work with Schema containers describing the entire SmartPlant data model (based on the EFSchema.xml file).

Schema objects include specific application objects that are implemented as *ClassDefs*, *InterfaceDefs*, *RelDefs*, etc. A schema container holds information such as the *name*

and unique identifier (*UID*) of a ClassDef. For example, the *PIDInstrument* class definition in a schema container might look like this:

```
<ClassDef>
  <IObject UID="PIDInstrument" Name="PIDInstrument"
           Description="P&&ID Instrument" />
  <IClassDef />
  <ISchemaObj DisplayName="Instrument"
           SchemaRevVer="03.06.01.24" />
</ClassDef>
```

Data Containers

Data containers hold information about specific instances of objects defined by the schema. For example, one particular instance of a PIDInstrument might look like this:

```
<PIDInstrument>
   <IObject UID="E2C252259F0A4596969BAFED07D5534A" Name="TT-
  12809" />
   <IInstrument />
   <IInstrumentOcc InstrumentLocation="@EE31D" />
   <IPBSItem ConstructionStatus="@NewConstruction"
  ConstructionStatus2="@{78398AB4-9F3D-11D6-BDA7-00104BCC2B69}"
  />
   <IPlannedMatl />
   <IDrawingItem />
   <IPipingPortComposition />
   <IHeatTracedItem HTraceRqmt="" />
   <ILoopMember />
   <IPartOcc />
   <IPressureReliefItem />
   <IProcessPointCollection />
   <ISignalPortComposition />
   <IDocumentItem />
   <IElecPowerConsumer />
   <IPart />
   <INoteCollection />
   <INamedInstrument InstrFuncModifier="T" InstrLoopSuffix=""
  InstrTagPrefix="Unit1" InstrTagSequenceNo="12809" InstrTagSuffix=""
  MeasuredVariable="T" />
 </PIDInstrument>
```

All containers implement the *IContainer* interface, defined in the *SchemaCompInterfaces* type library (…\Program Files\Common Files\Intergraph\Schema Component\SchemaComponent.tlb).

This type library is identified by the following name from *Project References* in VB:

Intergraph - EF/SC Component Interfaces

The *IContainer* interface has the following properties and methods:

Property	Description
CompSchema As String	Sets/returns the component schema for the objects within the container, e.g., PIDComponent. (If the container is XML file-based, then this will be an attribute of the container XML tag.)
ConnectionString As String	Returns the connection string used to connect to the data source for the container. (If the container is XML file-based, then the connection string will be empty.)
ContainedObjects As IObjectCollection	Returns the collection of objects in this container.
ContainerComposition As IContainerComposition	Sets (only once)/returns the container composition object that includes this container. Every container belongs to one, and only one, container composition.
IContainerQuery As IContainerQuery	Returns the implied *IContainerQuery* interface.
IsRepository As Boolean	Reserved for future use.
IsTransactionBased As Boolean	Returns whether the container updates after each transaction or needs to be saved.
Name As String	Sets (only once)/returns the name of the container. (If the container is XML file-based, then this will return the full path and file name of the XML file.)
SoftwareVersion As String	Sets/returns the software version for the container. (If the container is XML file-based. then this will be an attribute of the Container XML tag.)

Method	Description
AddObject(oObjIObj As IObject)	Adds the specified object to the container.
Close()	Closes the container.
CreateProxyObjectsForUnresolvedRels()	Creates proxy objects for all dangling relationship ends. (This is used if the container doesn't contain both objects at the ends of a relationship object.)
New(sFilename As String, eScope As eScopes, oCompIContainerComposition As IContainerComposition, [sConnectionString As String])	Creates a new container, specifies the scope of objects for the container, and ties it to the specified container composition.
Open(sFilename As String, oCompIContainerComposition As IContainerComposition, [sConnectionString As String])	Opens the container corresponding to the specified file name and ties it to the specified container composition.
RemoveObject(oObjIObj As IObject)	Removes the specified object from the container.
ResetContainer()	Sets container back to starting state.
ResolveUnresolvedRels()	Resolves all unresolved relationships in the container. This method is useful if relationships were loaded into the container before the objects they relate were loaded.
ResolveUnresolvedSharedProxies()	Resolves all unresolved shared proxies within the container.
Save()	Saves the contents of the container.
SaveAs(oSourceIContainer As IContainer, sFilename As String, oCompIContainerComposition As IContainerComposition, [sConnectionString As String])	Saves the contents of the container to a new file.

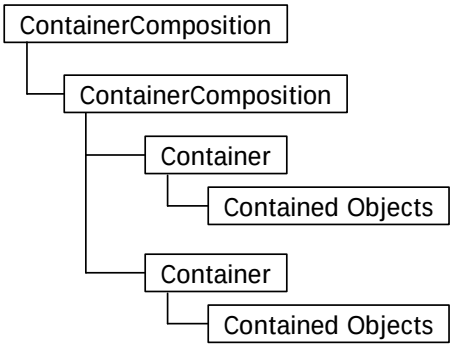
Containers may implement additional interfaces depending on their type. For example, an XML file-based container will also implement the IXMLContainer interface. The additional container interfaces are:

- **ICompSchemaContainer**
- **IMsgContainer**
- **ICorrelContainer**
- **IinstructionsContainer**
- **ISoftwareContainer**
- **IXMLContainer**
- **IXMLContainer2**

You can view the properties and methods for these additional container interfaces using the *Visual Basic Object Browser*.

3.2. ContainerCompositions

A *Container Composition* is an object that scopes (or owns) containers (similar to the folder/file relationship). A container may be owned by only one container composition; however, a container composition may scope multiple containers. A container composition may also scope other container compositions.



Container compositions implement the *IContainerComposition* interface with the following properties and methods:

Property	Description
ContainedContainers As IContainerCollection	Returns the collection of containers that have been assigned to this container composition. Every container should be assigned to one, and only one, container composition.
IContainer As IContainer	Returns a pointer to the implied *IContainer* interface. **Note**: a container composition exposes this interface directly for itself; it is not exposing one of its containers.
MessageContainer As IMsgContainer	Sets/returns the message container for this container composition.

Method	Description
GenerateUID() As String	Generates a globally unique identifier (GUID).
GetInstancesForClassDef(sClassDefUID As String) As IObjectCollection	Returns a collection of all objects with class definition *sClassDefUID* that exist in any container assigned to the container composition.

Method	Description
GetIObjectForUID(sUID As String) As IObject	Returns a pointer to the *IObject* interface for the object with the unique identifier *sUID*. For this method to succeed, the object must have been assigned to the container composition by a previous call to *SetIObjectForUID*.
ReleaseContents()	Releases all of the containers and objects within those containers that have been assigned to the container composition.
RemoveUID(sUID As String)	Removes the object with the specified UID from the container composition's UID/object collection.
SetIObjectForExplicitUID (oObjIObj As IObject, sUID As String)	Adds the object to the container composition's UID/object collection
SetIObjectForUID(oObjIObject As IObject)	Identifies the specified object as belonging to this container composition. This method is called automatically whenever the container for an object is identified (Container property on *IObject* interface).

3.3. IContainerHelper

The *IContainerHelper* interface provides methods to create and populate containers and container compositions. When you open files using the IContainerHelper methods, be sure to create a container composition for each type of container that is to be opened. Each container composition should be parented by the appropriate *ContainerComposition*. For example, the *ContainerComposition* for schema *Containers* should be parented by the meta schema *ContainerComposition*, and so on.

The IContainerHelper interface has the following methods:

Method	Description
CreateAndLoadMetaSchema(oMetaCompIContainerComposition As IContainerComposition) As Boolean	Creates the container composition for the meta schema, creates the containers for the meta schema objects and messages, and then loads the meta schema.
CreateCommonSchemaContainerComposition(oMetaCompIContainerComposition As IContainerComposition, oCommonCompIContainerComposition As IContainerComposition)	Creates a container composition for the common schema objects and ties it to the meta schema container composition.
CreateCommonSchemaContainers(oCommonCompIContainerComposition As IContainerComposition, oCommonContIContainer As IContainer)	Creates a container for the common schema objects and ties that container to the specified container composition.
CreateDocumentContainerComposition(oSchemaCompIContainerComposition As IContainerComposition, oDocCompIContainerComposition As IContainerComposition)	Creates a document container composition and ties it to the specified schema container composition.
CreateDocumentContainers(oDocCompIContainerComposition As IContainerComposition, oDocContIContainer As IContainer, oDocMetadataIContainer As IContainer, oDocTombstonesIContainer As IContainer)	Creates containers for a document, its meta data, and its tombstones and ties them to the document's container composition.

Method	Description
CreateDocumentContainersNoTombstones(oDocCompIContainerComposition As IContainerComposition, oDocContIContainer As IContainer, oDocMetadataIContainer As IContainer)	Creates the document container and document meta data container and ties them to specified document container composition.
CreateMovedInstructionsContainer(oDocCompIContainerComposition As IContainerComposition, oMovedInstructionsIContainer As IContainer)	Creates a container for MovedInstruction objects and ties it to the specified container composition.
CreateProxyObjectsForUnresolvedRels(oContIContainer As IContainer)	Creates proxy objects for all of the unresolved relationship ends in the specified container.
CreateSchemaContainerComposition(oCommonCompIContainerComposition As IContainerComposition, oSchemaCompIContainerComposition As IContainerComposition)	Creates a container composition for containing schema objects and ties it to the specified container composition
CreateSchemaContainers(oSchemaCompIContainerComposition As IContainerComposition, oSchemaContIContainer As IContainer)	Creates a container for containing schema objects and ties it to the specified schema container composition
CreateTombstoneContainer(oDocCompIContainerComposition As IContainerComposition, oDocTombstonesIContainer As IContainer)	Creates the document tombstones container and ties it to document's container composition.
CreateToolSchemaContainerComposition(oSchemaCompIContainerComposition As IContainerComposition, oToolSchemaCompIContainerComposition As IContainerComposition)	Creates a container composition for the tool schema and ties that container composition to the specified schema container composition.
CreateToolSchemaContainers(oToolSchemaCompIContainerComposition As IContainerComposition, oToolSchemaContIContainer As IContainer)	Create a container for the tool schema objects and ties it to the specified container composition.

Method	Description
LoadComponentSchema(sSchemaXMLFilename As String, oSchemaCompIContainerComposition As IContainerComposition, oSchemaContIContainer As IContainer) As Boolean	Loads the schema objects within the specified XML file into the specified schema container
RecreateComponentSchemaContainers(oSchemaCompIContainerComposition As IContainerComposition, oSchemaContIContainer As IContainer)	Releases and recreates the schema container
RecreateDocumentContainers(oDocCompIContainerComposition As IContainerComposition, oDocContIContainer As IContainer, oDocMetadataIContainer As IContainer, oDocTombstonesIContainer As IContainer)	Releases and recreates the containers for the document, document metadata and document tombstones
RecreateDocumentContainersNoTombstones(oDocCompIContainerComposition As IContainerComposition, oDocContIContainer As IContainer, oDocMetadataIContainer As IContainer)	Recreates the document and document metadata containers
ResolveUnresolvedRels(oContIContainer As IContainer)	Resolves all unresolved relationships in the container. This method is useful if relationships were loaded into the container before the objects they relate were loaded.
ResolveUnresolvedSharedProxies(oContIContainer As IContainer)	Resolves all unresolved shared proxies in the container.
RetrieveDocument(sDocumentXMLFilename As String, sDocTombstonesXMLFilename As String, oDocCompIContainerComposition As IContainerComposition, oDocContIContainer As IContainer, oDocTombstonesIContainer As IContainer) As Boolean	Loads the document and document tombstone containers with objects from parsing the specified XML files

Visual Basic Example Using IContainerHelper

Here is a small Visual Basic example that demonstrates creating and loading *Containers* and *ContainerCompositions* using the container helper object, *IContainerHelper* interface.

Note: *You must reference the Schema Component Interfaces in your Visual Basic project to be able to use any of the types or interfaces in the Schema Component or SmartPlant Integration APIs*

This type library is identified as:

Intergraph - EF/SC Component Interfaces

In the Visual Basic Project References dialog.

You must attach the Component Schema library to your Visual Basic project to create Helper objects. This library is named CompSchemaCont.dll and is located in the Program Files\Common Files\Intergraph\Schema Component directory.

This type library is identified as:

Intergraph - EF/SC Component Schema (CompSchemaCont)

in the Visual Basic Project References dialog.

You must attach the XMLComp library to your Visual Basic project to create an XMLContainer. This library is named XMLComp.dll and is located in the Program Files\Common Files\Intergraph\Schema Component directory.

This type library is identified as:

Intergraph - EF/SC XML Container (XMLComp)

in the Visual Basic Project References dialog.

Note: *To avoid the trouble of prefixing every interface or type in these libraries with the full namespace, you need to add an* **imports** *statement for each library at the top of each source file in which it will be used. (In C#, this is the* **using** *statement.)*

An alternative in VB is to create imports at the project level. In the references tab of the project properties, you can select libraries to be imported at the project level. ***This is a recommended best practice****.*

```
Dim oHelper As IContainerHelper
Dim oMetaCC As IContainerComposition
Dim oSchemaCC As IContainerComposition
Dim oSchemaContainer As IContainer

'// Create the Container Helper
Set oHelper = New Helper

'// Initialize the Meta Schema
oHelper.CreateAndLoadMetaSchema oMetaCC

'// Initialize the Schema ContainerComposition
oHelper.CreateCommonSchemaContainerComposition oMetaCC, oSchemaCC

'// Create and load the Schema Container from XML File
Set oSchemaContainer = New XMLContainer
oSchemaContainer.Open(My.Application.Info.DirectoryPath & "\PIDComponent.xml",
oSchemaCC)
```

3.4. IContainerHelper2

The *IContainerHelper2* interface has a single method, *CreateAllContainersAndContainerCompositions*, to perform the work of creating *Containers* and *ContainerCompositions* and filling them with data all in one step. This method provides the simplest way to load and initialize *Containers* and *ContainerCompositions.* This method only works for schema, tool schema, and data files that are XML files (*.xml).

Method	Description
CreateAllContainersAndContainerCompositions (bLoadInstructionsMetaSchema As Boolean, oMetaCompIContainerComposition As IContainerComposition, oSchemaContIContainer As IContainer, oToolSchemaContIContainer As IContainer, oDataContIContainer As IContainer, [sSchemaFilename As String], [sToolSchemaFilename As String], [sDataFilename As String])	Creates the containers and container compositions for the specified files and then parses those files.

Visual Basic Example Using IContainerHelper2

Here is a small Visual Basic example that demonstrates creating and loading *Containers* and *ContainerCompositions* using the container helper object, IContainerHelper2 interface.

```
Dim oHelper As IContainerHelper
Dim oHelper2 As IContainerHelper2
Dim oSchemaContainer As IContainer
Dim oDataContainer As IContainer
Dim oMetaCC As IContainerComposition

'// Create the Container Helper
Set oHelper = New Helper

'// Reference the ContainerHelper2 by accessing the IContainerHelper2 Interface
Set oHelper2 = oHelper

'// Use the Helper2.CreateAllContainersAndContainerCompositions method to
'// populate the Containers and ContainerCompositions from XML files
Call oHelper2.CreateAllContainersAndContainerCompositions(True, _
       oMetaCC, oSchemaContainer, Nothing, oDataContainer, _
       App.Path & "\PIDComponent.xml", "", App.Path & "\Data.xml")
```

3.5. Releasing ContainerComposition Objects

The *Schema Component* maintains many two-way object pointers. The reference count for these objects will, by default, never go to zero. Therefore, an explicit call must be made to release the objects in the container compositions before ending the application. Failure to do so will result in an error message being displayed.

All of the objects in a container composition are released when the *ReleaseContents* method for that container composition is called.

The *ReleaseContents* method should be called once for each container composition and should be called in the reverse order from which the container compositions were created. The following code illustrates this process for an example program in which meta schema, schema, and data container compositions were created.

```
'// Release the Container Composition contents
oDataCC.ReleaseContents
oSchemaCC.ReleaseContents
oMetaCC.ReleaseContents

Set oDataCC = Nothing
Set oSchemaCC = Nothing
Set oMetaCC = Nothing
```

Since most applications separate the initialization from the termination, the container composition pointers should be Private to the application rather than declared within a particular method (as shown in the previous examples).

3.6. Lab Exercise

Write a Visual Basic program to create and load a data container and schema container and print the number of objects in each container to the Debug window.

- Use the IContainerHelper2 code snippet above as a guide, open the meta schema, schema, and data containers using CreateAllContainersAndContainerCompositions.
- Print the count of objects from the meta schema container.
- Print the count of objects from the schema container.
- Print the count of objects from the data container.

You may use the Chapter 3 – Lab1 Folder as a starting point.

The completed lab exercise is available in the *Solutions* project folder.

4. Retrieving Objects in Containers

4.1. IObject

Every object held within a *Container* implements the *IObject* interface. This interface provides standard UID, Name, and Description properties as well as properties to access the class definition of the object, the *IClassDefComponent* interface, relationship collections, and so on.

Property	Description
ClassDefIObj As IObject	Sets/returns the class definition for the object (for example, the type of the object).
ContainedObject As IContainedObject	Sets/returns the contained object for the object. For certain containers a second object is contained in order to handle container-specific object functionality.
Container As IContainer	Sets/returns the container for the object. Every object with an IObject interface must belong to one, and only one, container.
Description As String	Optional property. Sets/returns the Description for the object.
DescriptionSetFlag As Boolean	Indicates whether Description has been set.
End1IRelCollection As IRelCollection	Returns the collection of relationships *for which this object is end 1*.
End2IRelCollection As IRelCollection	Returns the collection of relationships *for which this object is end 2*.
IClassDefComponent As IClassDefComponent	Returns the implied *IClassDefComponent* interface.
Name As String	Optional property. Sets/returns the Name for the object.
NameSetFlag As Boolean	Indicates whether Name has been set.
UID As String	Required property. Sets/returns the unique identifier for the object.

Property	Description
UIDSetFlag As Boolean	Indicates whether the UID has been set for the object.

Method	Description
Serialize() As String	Serializes the object's properties to a string.

4.2. ContainedObjects

Containers provide a property called *ContainedObjects* that returns a collection of IObjects within the container. Use the standard Visual Basic *For Each* loop to iterate the objects in the ContainedObjects collection. (See the Chapter 4 – Example 2 folder for a complete code listing.)

```
'// Using ContainedObjects on the Schema container
Dim oObject As IObject

For Each oObject In oSchemaContainer.ContainedObjects
    '// Do Something
Next
```

This allows you to serially process each object within the Container and access the IObject's properties and methods. For instance, you could replace the *'Do Something* label above to print the UID, Name, and Description properties.

```
'// Print Properties if set
With oObject
    If .UIDSetFlag Then Debug.Print "UID: "& .UID
    If .NameSetFlag Then Debug.Print "Name: "& .Name
    If .DescriptionSetFlag Then Debug.Print "Description: "& .Description
End With
```

If you are reading objects out of a data container, and you want to access or process only objects of a specific type, use the object's *ClassDefIObj* property to determine its type:

```
'// Using ContainedObjects on the Data container
Dim oObject As IObject

For Each oObject In oDataContainer.ContainedObjects
    If oObject.ClassDefIObj.UID = "PIDInlineInstrument" then
            '// Do Something
    End If
Next
```

Because the *ClassDefIObj* property returns the ClassDef or definition (type) from the schema, you can check its UID for a particular type. For example, one specific instance of an object from the data container may have the following UID:

`oObject.UID` returns "AAGGMOD179"

Check the object's ClassDefIObj.UID for the type or definition of the object:

`oObject.ClassDefIObj.UID` returns "PIDInlineInstrument"

You may also use the *IContainerQuery* interface to limit the types of objects returned from a container. See the next section for complete information on IContainerQuery.

4.3. IContainerQuery

All *Containers* provide an *IContainerQuery* interface to make finding specific objects or types of objects much easier. Use the *AllObjectsAreLoaded* property to determine if you need to call the *LoadAllObjects* method before using the query methods.

Property	Description
AllObjectsAreLoaded As Boolean	Indicates whether all objects for the container have been loaded into memory.

Method	Description
FindObjects(sClassDefUIDs As String, sUIDCriteria As String, sNameCriteria As String, sDescriptionCriteria As String, bCaseSensitive As Boolean, oObjsIObjectCollection As IObjectCollection)	Finds all objects that are in the list of class definition UIDs and that satisfy the UID/Name/Description criteria.
GetInstancesForClassDef(sClassDefUID As String, oObjsIObjectCollection As IObjectCollection)	Adds all objects of the specified class to the object collection.
GetInstancesForInterfacDef(sInterfaceDefUID As String, oObjsIObjectCollection As IObjectCollection)	Adds all objects having the specified interface to the object collection.
GetIObjectForUID(sUID As String) As IObject	Locates the object with the specified UID. May return Nothing.
LoadAllObjects()	Forces a load of all objects for the container into memory.

4.4. Get Objects with GetIObjectForUID

If the UID of an object is known, then locating that object simply involves a call to the *GetIObjectForUID* method. For example, to locate the ClassDef for the PIDInstrument object from a schema conatiner, the following code can be used (see the Chapter 4 – Example 3 folder for a complete code listing):

```
'// Using IContainerQuery on the Schema container
Dim oObject As IObject

Set oObject =
oSchemaContainer.IContainerQuery.GetIObjectForUID("PIDInstrument")
```

When retrieving a particular object from a data container, you must provide a UID for the instance:

```
'// Using IContainerQuery on the Data container
Dim oObject As IObject

Set oObject = oDataContainer.IContainerQuery.GetIObjectForUID("AAGGMOD179")
```

4.5. IObjectCollection

The *GetInstancesForClassDef* and *FindObjects* methods (covered in the following two sections) on the IContainerQuery interface return a collection of type *IObjectCollection*. You must create this type of collection using the New syntax provided by VB. The object type (*ObjectCollection*) that is Newed exists in the following typelib:

…\Program Files\Common Files\Intergraph\Schema Component\CompIntfcProp.dll

which displays with the following name from Project->References:

Intergraph - EF/SC Component Interface Implementations (IntfcProp)

You must attach this type library before Newing an IObjectCollection type variable and subsequently calling the GetInstancesForClassDef or FindObjects methods. The IObjectCollection interface includes the following properties and methods:

Property	Description
Count As Long	Returns the number of objects in the object collection.
Item(lIndex As Long) As IObject	Returns the object corresponding to the index/UID
NewEnum As IEnumVARIANT	Returns an enumerator for the object collection. (Basically means the collection supports For Each syntax.)

Method	Description
Add(oObjIObj As IObject)	Adds the object to the collection of objects
GetRelForDefUID(sDefUID As String, bUseEnd1Collections As Boolean, bForward As Boolean) As IRel	Returns the first relationship of the specified type for an object in the collection for the specified end and beginning with either the first or last object in the collection
GetRelsForDefUID(sDefUID As String, bUseEnd1Collections As Boolean, bForward As Boolean) As IRelCollection	Returns all relationships of the specified type for objects in the collection for the specified end and beginning with either the first or last object in the collection

Method	Description
Remove(lIndex As Long)	Removes the object corresponding to vIndexKey from the object collection

4.6. Get Objects with GetInstancesForClassDef

If you want to locate all objects of a specific type and you know the UID of the ClassDef, then you can use the GetInstancesForClassDef method to return a collection of objects to enumerate. For example, to locate all of the PIDInstrument objects, use the following code (see the Chapter 4 - Example 4 folder for a complete code listing):

```
'// Load an ObjectCollection of PIDInstruments
Dim collObjects As IObjectCollection

Set collObjects = New ObjectCollection
Call oDataContainer.IContainerQuery.GetInstancesForClassDef("PIDInstrument",
collObjects)
```

Once you have the object collection filled, you can enumerate it using a standard Visual Basic For Each loop using an IObject variable.

```
'// Using For Each to enumerate object collection
Dim oObject As IObject

For Each oObject In collObjects
    Debug.Print oObject.UID
Next
```

4.7. Get Objects with GetInstancesForInterfaceDef

If you want to locate all objects having a specific interface and you know the UID of the InterfaceDef, then you can use the GetInstancesForInterfaceDef method to return a collection of objects to enumerate. For example, to locate all instrument objects, use the following code (see the Chapter 4 - Example 4 folder for a complete code listing):

```
'// Load an ObjectCollection of instruments
Dim collObjects As IObjectCollection

Set collObjects = New ObjectCollection
Call oDataContainer.IContainerQuery.GetInstancesForInterfaceDef("IInstrument",
collObjects)
```

Once you have the object collection filled, you can enumerate it using a standard Visual Basic For Each loop using an IObject variable.

```
'// Using For Each to enumerate object collection
Dim oObject As IObject

For Each oObject In collObjects
    Debug.Print oObject.UID
Next
```

4.8. Get Objects with FindObjects

The FindObjects method allows you to specify criteria by which to select objects based on their ClassDef, UID, Name, or Description. This method supports wildcards (*). For example, to locate all of the Control Valve objects, use code similar to the following (see the Chapter 4 - Example 5 folder for a complete code listing):

```
'// Load an ObjectCollection of Control Valves
Dim collValves As IObjectCollection

Set collValves = New ObjectCollection
Call oDataIContainer.IContainerQuery.FindObjects("", "", "CV*", "", False,
collValves)
```

The following is another example of using FindObjects. This example uses the ClassDefUIDs parameter to specifiy two ClassDefs to search for: PIDInstrument and PIDInlineInstrument. Note carefully that the ClassDefs in this parameter must be delimited with the '~' character. This example will find all instances of those two ClassDefs in the container.

```
'// Load an ObjectCollection of all instruments
Dim collInstruments As IObjectCollection

Set collInstruments = New ObjectCollection
Call
oDataIContainer.IContainerQuery.FindObjects("PIDInstrument~PIDInlineInstrument
", "", "*", "", False, collInstruments)
```

Once you have the object collection filled, you can enumerate it using a standard Visual Basic For Each loop using an IObject variable as shown previously.

4.9. Lab Exercise

1. Loop through all the contained objects in the data container and print out the tag name of all instrument loops.
    a. Using the IContainer.ContainedObjects collection (data container), write a For Each statement to loop through all the objects.
    b. Test the ClassDef for 'PIDInstrumentLoop' using IObject.ClassDefIObj.Name.
    c. If the ClassDef is 'PIDInstrumentLoop', print out the tag name using IObject.Name.
2. Print out the tag number of the object having a UID of '573F431C6601454AB95733A3FEF6359C'.
    a. Using IContainer.IContainerQuery.GetIObjectForUID, find the object with the above UID and print out it's tag name.
3. Use IContainerQuery.GetInstancesForClassDef to print out the tag number of all instrument loops.
    a. Declare an IObjectCollection and initialize it with a new ObjectCollection.
    b. Using IContainer.IContainerQuery.GetInstancesForClassDef, get all the instances of "PIDInstrumentLoop".
    c. Write a For Each statement to loop through all and print out the tag name.
    d. Are your results the same as in #1 above?
4. Use IContainerQuery.GetInstancesForInterfaceDef to print out the tag number of all instrument loops.
    a. Declare an IObjectCollection and initialize it with a new ObjectCollection.
    b. Using IContainer.IContainerQuery.GetInstancesForInterfaceDef, get all the instances of "IInstrumentLoop".
    c. Write a For Each statement to loop through all and print out the tag name.
    d. Are your results the same as in #1 and #3 above?
5. Use IContainerQuery.FindObjects to print out the tag number of all instruments or loops starting with 'T*'.
    a. Declare an IObjectCollection and initialize it with a new ObjectCollection.
    b. Using IContainer.IContainerQuery.FindObjects, to find all instruments and loops starting with T*. Pass in three ClassDefs to search: PIDInstrument,

PIDInlineInstrument, and PIDInstrumentLoop. Pass in 'T*' as the name criteria.

c. Write a For Each statement to loop through all and print out the tag name.

d. Are your results the same as in #1, #3, and #4 above?

You may use the Chapter 4 – Lab 1 Folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

4.10. IClassDef

You have seen how to create and populate containers and how to locate objects contained within those containers using ContainedObjects and methods on the IContainerQuery interface. Objects returned support the IObject interface, which provides very generic information about an object but doesn't tell us what types of relationships the object is involved in.

The *IClassDef* interface exposes various relationships for an object, for example the interfaces an object *Realizes*. In addition it includes a *CreateInstance* method for creating objects of the specified type.

All objects in a schema container by definition are ClassDef objects and support the *IClassDef* interface. A data object from a data container may access its ClassDef counterpart via the ClassDefIObj property on IObject.

IClassDef supports the following Properties and Methods:

Property	Description
ClassFactoryDef As IRel	Returns the Instantiates relationship for which this object is end 2. Class factory definition (realizes IClassFactoryDef) for this relationship may be accessed using UID1.
ComponentSchema As IRel	Returns the Componentization relationship for which this object is end 2. Component schema (realizes ICompSchema) for this relationship may be accessed using UID1.
DeleteShared As Boolean	Optional property. Sets/returns whether creating a tombstone for an object of this class will terminate the shared object even if it was created by a different component.
DeleteSharedSetFlag As Boolean	Returns whether the DeleteShared property has been set.
IMappableClass As IMappableClass	Returns a pointer to the implied IMappableClass interface.
Instances(oCompIContainerComposition As IContainerComposition) As IObjectCollection	Returns collection of objects in the specified container composition that are instances of this class definition.
ISchemaObj As ISchemaObj	Returns a pointer to the implied ISchemaObj interface.

Property	Description
ISoftwareImplementable As ISoftwareImplementable	Returns a pointer to the implied ISoftwareImplementable interface.
ModelForClass As IRel	Returns the ModelClass relationship for which this object is end 2. Model definition (realizes IModelDef) for this relationship may be accessed using UID1.
PrimaryInterfaceDef As IRel	Returns the PrimaryInterface relationship for which this object is end 1. Interface definition (realizes IInterfaceDef) that is the primary interface may be accessed using UID2.
PropertyDefsForRealizedInterfaces As IObjectCollection	Returns the collection of property definitions (realize IPropertyDef) that end at an interface realized by this class definition.
RealizedInterfaceDefs As IRelCollection	Returns the collection of Realizes relationships for which this object is end 1. Realized interface definitions (realize IInterfaceDef) may be accessed using UID2 for each relationship in collection.
RelDefsForRealizedInterfaces As IObjectCollection	Returns the collection of relationship definitions (realize IRelDef) that end at an interface realized by this class definition. May return Nothing.
SharedObjDefs As IRelCollection	Returns the collection of Sharing relationships for which this object is end 2. Shared object definitions (realize ISharedObjDef) may be accessed using UID1 for each relationship in collection. May return Nothing.

Method	Description
AddInterfaceDefsUsedByRelDefsToObject(oObjIObj As IObject)	Adds the interfaces corresponding to the interface definitions realized by this class definition that are involved in a relationship definition to the specified object.
CanClassDefBeShared(sClassDefUID As String) As Boolean	Tests whether the specified ClassDef has a Sharing relationship with a shared object definition that is also related to this class definition.
CreateInstance(oContainerIContainer As IContainer) As IObject	Creates an object of this class definition type and adds it to the specified container.
GetInterfaceDefsUsedByRelDefs() As IObjectCollection	Returns a collection of all the interface definitions (realize IInterfaceDef) realized by this class definition that are involved in a relationship definition.

You can use the *RealizedInterfaceDefs* property to enumerate interfaces realized by the ClassDef. You can access an object's IClassDef interface by declaring a variable of the proper type and setting it equal to the IObject object.

```
'// Get the IClassDef interface
Dim oClassDef As IClassDef

'// For an object from data container
Set oClassDef = oObject.ClassDefIObj
```

Because the RealizedInterfaceDefs property is a collection of relationships, you need an *IRel* object to enumerate those interfaces *(relationships are discussed in detail in Chapter 7).* You can use a standard Visual Basic For Each loop to access the items in the collection (see the Chapter 4 - Example 6 folder for a complete code listing).

```
'// Using RealizedInterfaceDefs to enumerate interfaces
Dim oRel As IRel

For Each oRel In oClassDef.RealizedInterfaceDefs
    Debug.Print oRel.UID2IObj.UID
Next
```

4.11. Lab Exercise

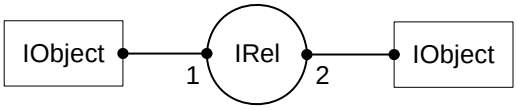
1. Write a program to collect and print a list of all the ClassDefs represented in the sample data container.
    a. Create a standard collection to hold strings.
    b. Enumerate through all the objects in the data container.
    c. If the object ClassDef is not already in the ClassDef collection, add it.
    d. Print all the ClassDefs from the collection.
2. Also print out the number of Realizes from each of the collected ClassDefs.
    a. For each of those ClassDefs, get a reference using IContainerQuery.GetIObjectForUID.
    b. Get a reference to the IClassDef interface.
    c. Print out the number of Realizes from IClassDef.RealizedInterfaceDefs.Count.
3. Did anything surprise you about the output?

You may use the Chapter 4 – Lab 2 Folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

5. Traversing Relationships

A relationship (IRel) connects two objects together. On a schema level, a relationship connects two interfaces together; on a data level; a relationship connects two objects together.



5.1. IRel and IRelCollection

The *IObject* interface exposes two collections of IRel objects: *End1RelCollection* and *End2RelCollection*. The data type for these collections is *IRelCollection*. End1IRelCollection contains all of the relationships where the IObject is End1 of a relationship; End2IRelCollection contains all of the relationships where the IObject is End2 of a relationship.

If you review the xml representation for an IRel object, it's easier to understand the terminology used in the automation model—here is a subset of the End2IRelCollection for a PIDProcessEquipment instance whose UID is "AAGGMOD50003"):

```
<Rel>
  <IObject UID="AAGGMOD50004MOD50003" />
  <IRel UID1="AAGGMOD50004" UID2="AAGGMOD50003"
DefUID="EquipmentComponentComposition" />
  </Rel>
<Rel>
  <IObject UID="AAGGMOD50006MOD50003" />
  <IRel UID1="AAGGMOD50006" UID2="AAGGMOD50003"
DefUID="EquipmentComponentComposition" />
  </Rel>
<Rel>
  <IObject UID="AAGGREP50006MOD50003" />
  <IRel UID1="AAGGREP50006" UID2="AAGGMOD50003" DefUID="DrawingItems" />
</Rel>
```

Each IRel object has its own IObject UID, just like every other object (the model chosen here is to concatenate the UIDs of the two involved objects). There are three other elements of the relationship:

1) the UID for the object at End1 (UID1),

2) the UID for the object at End2 (UID2), and

3) the name of the relationship (DefUID).

Thus, you can think of an *IRelCollection* (either End1IRelCollection or End2IRelCollection) of IRel objects as looking something like the following table:

UID1	DefUID	UID2
AAGGMOD50004	EquipmentComponentComposition	AAGGMOD50003
AAGGMOD50006	EquipmentComponentComposition	AAGGMOD50003
AAGGREP50006	DrawingItems	AAGGMOD50003

The *IRel* interface has the following properties and methods:

Property	Description
DefUID As String	Optional property (expected to be set except for referenced relationship definitions). Sets/returns the unique identifier for the relationship definition for the relationship.
DefUIDIObj As IObject	Sets/returns the relationship definition for the relationship.
DefUIDSetFlag As Boolean	Indicates whether the *DefUID* property has been set.
IMapRel As IMapRel	Returns the optionally implied *IMapRel* interface. May return Nothing.
IncludeInCompSchema As Boolean	Not currently used. Sets/returns whether the relationship should be included in a component schema extracted from this schema.
ISchemaObj As ISchemaObj	Returns the optionally implied *ISchemaObj* interface. May return Nothing.
IsRequired As Boolean	Optional property. Returns/sets whether the relationship is required or whether it is optional.
IsRequiredSetFlag As Boolean	Indicates whether the *IsRequired* property has been set.
OrderValue As Long	Optional property. Sets/returns the numeric value used for ordering relationships.
OrderValueSetFlag As Boolean	Indicates whether the OrderValue has been set.

Property	Description
UID1 As String	Required property. Sets/returns the unique identifier for the object at end 1 of the relationship.
UID1IObj As IObject	Sets/returns the object at end 1 of the relationship.
UID1SetFlag As Boolean	Indicates whether the UID1 property has been set.
UID2 As String	Required property. Sets/returns the unique identifier for the object at end 2 of the relationship.
UID2IObj As IObject	Sets/returns the object at end 2 of the relationship.
UID2SetFlag As Boolean	Indicates whether the UID2 property has been set.

Method	Description
UpdateUID1UsingUID1IObj()	Explicitly updates UID1 from its object. Only called if UID for referenced object changes.
UpdateUID2UsingUID2Iobj()	Explicitly updates UID2 from its object. Only called if UID for referenced object changes.

You can use the IRel objects in an IRelCollection to navigate from one interface (or object) to another interface (or object). The *UID1IObj* and *UID2IObj* properties allow you to locate the connected object on the other side.

In the *IClassDef* example from the previous chapter, you saw how to enumerate the interfaces that an object *realizes* by navigating relationships at the schema level. You can do the same thing at the data level as shown in the following example (see the Chapter 5 - Example 7 folder for a complete code listing):

```
'// Create an Object Collection
Dim collEquipment As IObjectCollection
Dim oObject As IObject
Dim oRel as IRel

Set oIObjectCollection = New ObjectCollection

'// Load the collection with PIDProcessEquipment data
Call
oDataContainer.IContainerQuery.GetInstancesForClassDef("PIDProcessEquipment",
collEquipment)

'// Make sure we got an object
If collEquipment.Count > 0 Then

    '// Get a single object
    Set oObject = collEquipment.Item(1)

    '// Loop through all of the Relationships for which this object is End2
    For Each oRel In oObject.End2IRelCollection
        Debug.Print oRel.UID1IObj.Name
    Next
End If
```

There are several points to make:

1. You must be familiar with the data model (Schema) and how relationships connect objects to be able to navigate.

2. The end (UID1 or UID2) of the relationship is crucial.

3. If you are dealing with End1 relationships, navigate to UID2IObj to get to the object on the other side.

4. If you are dealing with End2 relationships, navigate to UID1IObj to get to the object on the other side.

5.2. IRelDef

Just like schema objects implement the *IClassDef* interface, relationships implement an *IRelDef* interface that you can use to access information about the relationship. You can access the IRelDef interface through the IRel.DefUIDIObj property. The IRelDef interface has the following properties and methods:

Property	Description
End1Locality As eLocalities	Required property. Sets/returns the locality of reference for end 1 of the relationship definition.
End1LocalitySetFlag As Boolean	Indicates whether End1Locality has been set.
End2Locality As eLocalities	Required property. Sets/returns the locality of reference for end 2 of the relationship definition.
End2LocalitySetFlag As Boolean	Indicates whether End2Locality has been set.
IRel As IRel	Returns the implied *IRel* interface.
IRelDef2 As IRelDef2	Returns the optionally implied *IRelDef2* interface. May return Nothing.
IsAbstract As Boolean	Optional property. Sets/returns whether relationship definition is abstract or concrete.
IsAbstractSetFlag As Boolean	Indicates whether IsAbstract has been set.
MapPropertyToRelDefMapProperties As IRelCollection	Returns collection of MapPropertyToRelDef relationships for which this object is end 2. Map property definitions (realize IMapPropertyDef) may be accessed using UID1 for each relationship. May return Nothing.

Property	Description
MapRelDefToRelDefMapRelDefs As IRelCollection	Returns collection of MapRelDefToRelDef relationships for which this object is end 2. Map relationship definitions (realize IMapRelDef) may be accessed using UID1 for each relationship. May return Nothing.
Max1 As Integer	Required property. Sets/returns the maximum cardinality at end 1 of the relationship definition. A value of zero indicates to-many (*).
Max1SetFlag As Boolean	Indicates whether the Max1 property has been set.
Max2 As Integer	Required property. Sets/returns the maximum cardinality at end 2 of the relationship definition. A value of zero indicates to-many (*).
Max2SetFlag As Boolean	Indicates whether the Max2 property has been set.
Min1 As Integer	Required property. Sets/returns the minimum cardinality at end 1 of the relationship definition.
Min1SetFlag As Boolean	Indicates whether the Min1 property has been set.
Min2 As Integer	Required property. Sets/returns the minimum cardinality at end 2 of the relationship definition.
Min2SetFlag As Boolean	Indicates whether the Min2 property has been set.
Ordering As eRelOrdering	Optional property. Defines the type of ordering to be applied to relationships for this relationship definition.
OrderingSetFlag As Boolean	Indicates whether Ordering is set.

Property	Description
RelDefToMapPropertyMapProperty As IRelCollection	Returns collection of RelDefToMapProperty relationships for which this object is end 1. Map property definitions (realize IMapPropertyDef) may be accessed using UID2 for each relationship. May return Nothing.
RelDefToMapRelDefMapRelDef As IRelCollection	Returns collection of RelDefToMapRelDef relationships for which this object is end 1. Map relationship definitions (realize IMapRelDef) may be accessed using UID2 for each relationship. May return Nothing.
Role1 As String	Required property. Sets/returns the role associated with end 1 of the relationship definition.
Role1SetFlag As Boolean	Indicates whether the Role1 property has been set.
Role2 As String	Required property. Sets/returns the role associated with end 2 of the relationship definition.
Role2SetFlag As Boolean	Indicates whether the Role2 property has been set.
SpecOfUID As String	Optional property. Sets/returns the unique identifier for the relationship definition for which this relationship definition is a specialization.
SpecOfUIDIObj As IObject	Sets/returns the relationship definition for which this relationship definition is a specialization.
SpecOfUIDSetFlag As Boolean	Indicates whether the SpecOfUID property has been set.
UsesMapping As Boolean	Optional property. Sets/returns whether relationships of this type set End1MapValue and/or End2MapValue.

Property	**Description**
UsesMappingSetFlag As Boolean	Indicates whether UsesMapping is set.

Method	Description
CreateInstance(oObjContIContainer As IContainer) As IObject	Creates a relationship and sets the DefUID for that relationship to this object.

You may use the *IRelDef* object to create new relationships, which is covered in the next chapter. There is also a method on *IRelHelper* for creating relationships, which is also covered.

5.3. IRelDef2

There is an additional interface called *IRelDef2* that you can access from the IRelDef interface. Not all IRelDef objects provide access to the IRelDef2 interface. If there is no implied IRelDef2, then the IRelDef2 property of the IRelDef object will return nothing. IRelDef2 has the following properties:

Property	Description
Copy12 As eCopyTypes	Optional property (Default=DontCopy). Sets/returns whether copying the object at end 1 copies the object at end 2, copies the relationship to the object at end 2, or doesn't copy either.
Copy12SetFlag As Boolean	Indicates whether Copy12 has been set.
Copy21 As eCopyTypes	Optional property (Default=DontCopy). Sets/returns whether copying the object at end 2 copies the object at end 1, copies the relationship to the object at end 1, or doesn't copy either.
Copy21SetFlag As Boolean	Indicates whether Copy21 has been set.
Delete12 As Boolean	Optional property (Default=False). Sets/returns whether the deletion of the object at end 1 of the relationship forces the deletion of the object at end 2.
Delete12SetFlag As Boolean	Indicates whether Delete12 has been set.
Delete21 As Boolean	Optional property (Default=False). Sets/returns whether the deletion of the object at end 2 of the relationship forces the deletion of the object at end 1.
Delete21SetFlag As Boolean	Indicates whether Delete21 has been set.
FilterEdge12 As Boolean	Optional property (Default=False). Sets/returns whether the edge for traversing from end 1 to 2 should be filtered out in the GUI display.
FilterEdge12SetFlag As Boolean	Indicates whether FilterEdge12 has been set.

Property	Description
FilterEdge21 As Boolean	Optional property (Default=False). Sets/returns whether the edge for traversing from end 2 to 1 should be filtered out in the GUI display.
FilterEdge21SetFlag As Boolean	Indicates whether FilterEdge21 has been set.
IObject As IObject	Returns the implied IObject interface.

5.4. IRelHelper

As with the *IContainerHelper* object, the *IRelHelper* object provides methods that you can use to perform in a single step what might take a series of steps using IRel.

Note: *In the following Descriptions, oRel is used to represent the relationship object(s) in the IRelCollection passed into the function (oIRelCollection).*

Method	Description
CreateRelationship(oIContainer As IContainer, sUID As String, vUID1, vUID2, vDefUID) As IRel	Creates a relationship between UID1 and UID2 of type DefUID and with specified UID.
GetRelForDefUID(oIRelCollection As IRelCollection, sDefUID As String) As IRel	Returns the first relationship in oIRelCollection where oRel.DefUID = sDefUID.
GetRelForDefUIDAndMapValues(oIRelCollection As IRelCollection, sDefUID As String, sEnd1MapValue As String, sEnd2MapValue As String) As IRel	Returns relationship that is of specified type and matches the specified End1 and End2 map values.
GetRelForDefUIDAndUID1(oIRelCollection As IRelCollection, sDefUID As String, sUID1 As String) As IRel	Returns the first relationship in oIRelCollection where oRel.DefUID = sDefUID and oRel.UID1 = sUID1. **Note** • This function should only be called using the End2 collection.
GetRelForDefUIDAndUID2(oIRelCollection As IRelCollection, sDefUID As String, sUID2 As String) As IRel	Returns the first relationship in oIRelCollection where oRel.DefUID = sDefUID and oRel.UID2 = sUID2. **Note** • This function should only be called using the End1 collection.

Method	Description
GetRelForUID1(oIRelCollection As IRelCollection, sUID1 As String) As IRel	Returns the first relationship in oIRelCollection where oRel.UID1 = sUID1. **Note** This function should only be called using the End2 collection.
GetRelForUID2(oIRelCollection As IRelCollection, sUID2 As String) As IRel	Returns the first relationship in oIRelCollection where oRel.UID2 = sUID2. **Note** • This function should only be called using the End1 collection.
GetRelsForDefUID(oIRelCollection As IRelCollection, sDefUID As String) As IRelCollection	Returns the collection of relationships from oIRelCollection where oRel.DefUID = sDefUID.
GetRelsForDefUIDAndMapValues (oIRelCollection As IRelCollection, sDefUID As String, sEnd1MapValue As String, sEnd2MapValue As String) As IRelCollection	Returns relationships that are of specified type and match the specified End1 and End2 map values
GetRelsForDefUIDAndUID1(oIRelCollection As IRelCollection, sDefUID As String, sUID1 As String) As IRelCollection	Returns the collection of relationships from oIRelCollection where oRel.DefUID = sDefUID and oRel.UID1 = sUID1. **Note** • This function should only be called using the End2 collection.
GetRelsForDefUIDAndUID2(oIRelCollection As IRelCollection, sDefUID As String, sUID2 As String) As IRelCollection	Returns the collection of relationships from oIRelCollection where oRel.DefUID = sDefUID and oRel.UID2 = sUID2. **Note** • This function should only be called using the End1 collection.

Method	Description
GetRelsForUID1(oIRelCollection As IRelCollection, sUID1 As String) As IRelCollection	Returns the collection of relationships from oIRelCollection where oRel.UID1 = sUID1. **Note** • This function should only be called using the End2 collection.
GetRelsForUID2(oIRelCollection As IRelCollection, sUID2 As String) As IRelCollection	Returns the collection of relationships from oIRelCollection where oRel.UID2 = sUID2. **Note** • This function should only be called using the End1 collection.

The IRelHelper functions provide various ways to filter the End1IRelCollection and End2IRelCollection relationship collections. For example, the code below shows how to use the *IRelHelper.GetRelsForDefUID()* method to get only "Realizes" relationships versus manually filtering for these relationships :

```
Dim oDrawing As IObject
Dim oRelHelper As IRelHelper
Dim collRels As IRelCollection
Dim oRel As IRel

Set oDrawing = oSchemaContainer.IContainerQuery.GetIObjectForUID("PIDDrawing")

'// ONE-Show IRelHelper approach to filtering collection
'// Create a helper
Set oRelHelper = New Helper

'// Get the relationship collection
Set collRels = oRelHelper.GetRelsForDefUID(oDrawing.End1IRelCollection,
"Realizes")

'// Loop through the collection
For Each oRel In collRels
    Debug.Print oRel.UID2IObj.UID
Next

'// TWO-Show manual approach to filtering collection
'// Loop through the collection
Set collRels = oDrawing.End1IRelCollection
For Each oRel In collRels
   If oRel.DefUIDIObj.Name = "Realizes" Then
          Debug.Print oRel.UID2IObj.UID
   End If
Next
```

If a helper function is not used, then the test within the code (If (oIRel.DefUID.Name = "Realizes") Then) needs to account for relationship specializations if they occur.

5.5. Lab Exercise

Write a VB program to find the UIDs of objects that are related to PIDDrawing. Check when PIDDrawing is the End1 side of the relationship as well as the End2 side.

1. Write a VB program to find the all the objects related to the PIDDrawing itself. Check both the End1 and End2 rels. Print the RelDef, ClassDef, UID, and Name for each related object.

    a. Use GetInstancesForClassDef, GetInstancesForInterfaceDef, or FindObjects to get a reference to the PIDDrawing.

    b. Enumerate through the end1 rels to print out the RelDef and the ClassDef, UID, and name from the end2 objects.

    c. Enumerate through the end2 rels to print out the RelDef and the ClassDef, UID, and name from the end1 objects.

2. Use IRelHelper.GetRelsForDefUID to find all the objects related with the DrawingRepresentationCollection rel.

    a. Use GetInstancesForClassDef, GetInstancesForInterfaceDef, or FindObjects to get a reference to the PIDDrawing.

    b. Use IRelHelper.GetRelsForDefUID to filter down to the DwgRepresentationComposition rels.

    c. Print the ClassDef, UID, and Name from each related item.

You may use the Chapter 5 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

6. Intrinsic versus Application Objects

There are two categories of objects that exist within the Schema Component:

1) Intrinsic objects defined by the MetaSchema
2) Application objects defined by the Schema (EFSchema.xml). This category comprises both definition objects from schema containers and data instances from data containers.

Intrinsic objects are defined in the MetaSchema and include *ClassDefs, InterfaceDefs, RelDefs*, etc.—data types that define the default roles an object can play within SmartPlant. The *IObject* interface is the ultimate example; all objects support the IObject interface in order to retrieve properties like Name and UID. These object types are delivered with the software and are not dynamic.

Application objects are defined in the Schema (EFSchema.xml) and include those classes, interfaces, relationships, etc. modeled with the Schema Editor. (In the examples, we use a data model from PIDComponent.xml, a subset of data that normally resides in EFSchema.xml.) These object types are dynamic because the user creates and edits their definitions, roles, etc. with the Schema Editor. Application objects are based on object types provided by the MetaSchema, i.e., ClassDef, InterfaceDef, RelDef, etc. Instances of these application objects exist in data containers.

The most important distinction to draw between these two categories is that the methods for accessing intrinsic interfaces and properties differ from accessing application-defined interfaces and properties.

6.1. Accessing Intrinsic Interfaces on Objects

To gain access to an intrinsic interface (e.g., IClassDef) for an object, you can simply Dim a variable of the appropriate type and set an IObject reference to the variable. (For those familiar with COM programming, this is analogous to performing a *QueryInterface* on the object.) In Chapter 5, you saw an example of this when getting the IClassDef interface for an object from a schema container:

```
Dim oObj As IObject
Dim oClassDef As IClassDef

'// Get IObject for PIDInstrument from the schema container
Set oObj = oSchemaContainer.IContainerQuery.GetIObjectForUID("PIDInstrument")

'// Get the IClassDef interface for the object
Set oClassDef = oObj
```

However, if you want to access an interface for PIDInstrument that is dynamically defined (e.g, *IInstrument*), you would use a different approach. You must use the *IClassDefComponent* interface. The next section details this approach.

6.2. Accessing Application Interfaces on Objects

To access an interface other than IObject on an application-defined object requires using the *GetInterface* method on the *IClassDefComponent* interface, defined below.

Property	Description
InterfaceCollection(bAddRequiredInterfaces As Boolean) As IInterfaceCollection	Returns the collection of interfaces (realize IInterface) for the object. If AddRequiredInterfaces is True, then any interfaces defined as required that are currently missing from the interface collection will be added.

Method	Description
GetInterface(sInterfaceDefUID As String, bAddIfMissing As Boolean) As IInterface	Returns the interface (realizes IInterface) for the specified UID (InterfaceDefUID). If AddIfMissing is True, then the interface specified by InterfaceDefUID will be added to the interface collection (if it is missing from that collection).

For example, to access the IConnector interface for an instance of PIDPipingConnector, the following logic is required:

```
Dim oObj As IObject
Dim oConnector As IInterface

'// Get a specific PIDPipingConnector from the data container
Set oObj =
oDataContainer.IContainerQuery.GetIObjectForUID("CC3960BA2F4B439ABF79D9C6302CB
B68")

'// Use the IClassDefComponent interface to call GetInterface()
Set oConnector = oObj.IClassDefComponent.GetInterface("IConnector", True)
```

Note: *Accessing application interfaces and properties is relevant when dealing with objects from data containers. Definition objects (ClassDefs) within schema containers have other methods for accessing InterfaceDefs and PropertyDefs.*

In other words, instances within data containers have Interfaces and Properties to retrieve. Definition objects within schema containers have InterfaceDefs and PropertyDefs that make up the abstract definition of an object but contain no real data.

6.3. Lab Exercise

1. Write a VB program that prints all of Interfaces on an instance of the PIDPipingConnection ClassDef named “P-14413-3"-PM41-” (in the data container).

    a. Use IContainerQuery.FindObjects to find the piping connector named “P-14413-3"-PM41-”.

    b. Get a reference to the IClassDefComponent interface.

    c. Using IClassDefComponent.InterfaceCollection, print out all the instanced interfaces.

2. Write VB program that prints all of the InterfaceDefs Realized by the ClassDef PIPPipingConnector (in the schema container).

    a. Using IContainerQuery.GetIObjectforUID, get a reference to the ClassDef PIPPipingConnector.

    b. Get a reference to the IClassDef interface.

    c. Using IClassDef.RealizedInterfaceDefs, print out all the realized interface defs.

3. Compare the output from #1 and #2 above. Can you explain the difference?

You may use the Chapter 6 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

6.4. Accessing Properties on Intrinsic Interfaces

You can set or retrieve property values on an intrinsic interface object by using the standard VB *Object.Property* convention. For example, to set the UID for an object, simply call *Object.UID* as shown below.

```
Dim oPipingComponent As IObject

'// Create the piping component object (logic not shown)
Set oPipingComponent = ...

'// Set the UID for the piping component
oPipingComponent.UID = "MyNewPipingComponent"
```

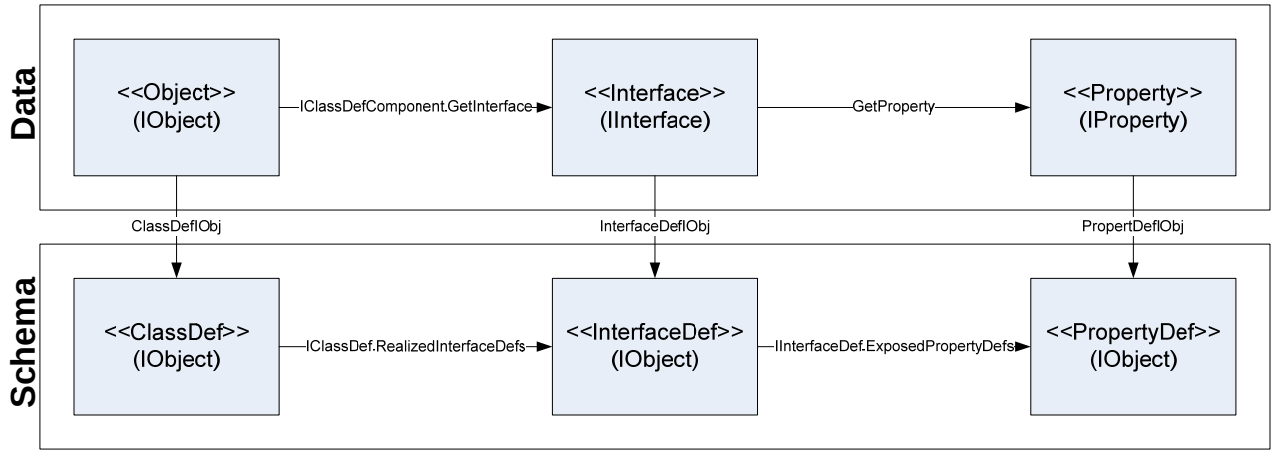
6.5. Accessing Properties on Application Interfaces

To access properties on an application-defined interface object requires using the *GetProperty* method on the *IInterface* interface, defined below.

Property	Description
InterfaceDefIObj As IObject	Returns the IObject interface for the interface definition for this interface.
PropertyCollection As IPropertyCollection	Returns the collection of properties for this interface.

Method	Description
GetProperty(sPropDefUID As String, bAddIfMissing As Boolean) As IProperty	Returns the IProperty for the property specified by sPropDefUID. If bAddIfMissing is True, then the property will be added to the interface if it currently is missing from that interface.
RemoveProperty(oPropToRemoveIProperty As IProperty)	Removes the specified property from the interface.

The following picture illustrates how you can use the APIs to move between objects, interfaces, and properties in data and the ClassDefs, InterfaceDefs, and PropertyDefs in schema.



For example, to access the FlowDirection property on the IConnector interface of an instance of PIDPipingConnector, the logic is as follows:

```
Dim oObj as IObject
Dim oConnector As IInterface
Dim oProperty As IProperty

'// Get a specific PIDPipingConnector from the data container
Set oObj = oDataContainer.IContainerQuery.GetIObjectForUID("AAGGMOD1773")

'// Use the IClassDefComponent interface to call GetInterface()
Set oConnector = oObj.IClassDefComponent.GetInterface("IConnector", True)

'// Get the PipingConnectorType property
Set oProperty = oConnector.GetProperty("FlowDirection", True)
```

Once you have retrieved an IProperty object, you will use its *Value* property to retrieve and set property values. IProperty is defined below:

Property	Description
IsValueNull As Boolean	Returns whether the property's value is Null.
PropertyDefIObj As IObject	Returns the IObject interface for the property definition for this property.
Value As String	Sets/returns the value for the property.
ValueSetFlag As Boolean	Indicates whether the value for the property has been set.

Method	Description
SetValueToNull()	Sets the property's value to Null.

All properties *of any type* support setting and retrieving values as a *String*. The *Value* property on IProperty is used to set the value for the property using a string or to return the property value as a string. If the property's type is *String*, then obviously, this is good enough.

You may choose to use this string and do your own conversion. For example, a property of type Boolean returns "True" if the value is True or "False" if the value is False. Therefore, an application can use its own parse logic to retrieve/set a Boolean property (shown below for IsInsulated).

```
Dim oIsInsulatedProp As IProperty

'// Get IsInsulated property (logic not shown)
Set oIsInsulatedProp = ...

If (oIsInsulatedProp.Value = "True") Then
    '// Do something
Else
    '// Do something else
```

For properties that are enumerated types, meaning they are scoped by *EnumListType* or *EnumListLevelType*, there are three ways to set the value:

```
1) oProperty.Value = "string"
2) oProperty.Value = "@UID"
3) oProperty.Value = "#(EnumNumber property value)"
```

Note: The third method is available only if the EnumNumber property has been set for the entries in the enumerated lists. This is not a requirement and you must check in the Schema Editor to verify the values of the EnumNumber property for the enumerated entries.

6.6. Lab Exercise

Write a VB program that retrieves and sets a property value. First, retrieve the *ProcFunc* property (exposed on the *IInstrument* interface) for an instance of a *PIDInstrument* from the data container. Then set the value to a different appropriate value and print that value out. Experiment with setting the value based on the string value, the UID value, or the index value from the *Entries* list.

1. Write a VB program that prints out the value of the ProcFunc property (exposed by the IInstrument InterfaceDef).

a. Get the first instance of an instrument using IContainerQuery.GetInstancesForInterfaceDef.

b. Get a reference to the IInstrument interface using IClassDefComponent.GetInterface.

c. Get a reference to the ProcFunc property using IInterface.GetProperty.

d. Print out the value.

2. Change the value of the ProcFunc property using the string value and then the UID value printing after each.

   a. Change the value to “Relief Device” and print out the value to verify.

   b. Change the value to “EE3F9”, which is the UID for “Relief Device” and print out the value to verify.

3. Did you notice that the value did not print out exactly as you set it?

You may use the Chapter 6 – Lab 2 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

6.7. Scalar Property Types

As you learned the modeling course, there are various scalar property types such as string, integer, double, date-time, Boolean, etc. So far, we've only treated properties as string using the IProperty interface; IProperty.Value is the string value from any property type.

There are, however, interfaces defined for all the scalar property types.

6.7.1. IIntProp Interface

For integral property types, the IIntProp interface is available…

Property	Description
IProperty As IProperty	Returns a reference back to the IProperty interface of this property.
Value As Integer	Gets or sets the integral value.

To use this interface for integral properties, for example…

```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty("MyIntProperty", True)

Dim  oIIntProp As IIntProp = oIProperty
oIIntProp = 125
```

Note that setting the string value on IProperty to a string representation of an integer will result in the string being parsed and the integer value will be set.

6.7.2. IDoubleProp Interface

For real property types, the IDoubleProp interface is available…

Property	Description
IProperty As IProperty	Returns a reference back to the IProperty interface of this property.
Value As Double	Gets or sets the floating point value.
ValueEnglishText As String	

ValueText As String

To use this interface for real properties, for example…

```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty("MyRealProperty", True)

Dim  oIDoubleProp As IDoubleProp = oIProperty
oIDoubleProp = 125.55
```

Note that setting the string value on IProperty to a string representation of an floating point number will result in the string being parsed and the integer value will be set. Standard and scientific notation are both parsed correctly.

6.7.3. IDateTimeProp Interface

For date-time property types, the IDateTimeProp interface is available…

Property	Description
Day As Short	The day of the month (1-31).
DaySI As Short	The day of the month in SI (GMT)
Hour As Short	The hour of the day (1-24).
HourSI As Short	The hour of the day in SI (GMT).
Millisecond As Short	The millisecond component of the time.
Minute As Short	The minute of the hour (0-59).
MinuteSI As Short	The minute in SI (GMT).
Month As Short	The month of the year (1-12).
MonthSI As Short	The month in SI (GMT).
Second As Short	The second of the minute (0-59).
SPFTextValue As String	The time as expressed in SPF date-time format.

TextValue As String	The date-time formatted as string. Identical to IProperty.Value.
Year As Short	The year component of the date-time.
YearSI As Short	The year in SI (GMT).
GetDeltaToDate() As Double	Calculates the difference between two dates.
SetValueToNow()	Set the value of this property to the current date/time.

To use this interface for date time properties, for example…

```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty("MyDateProperty", True)

Dim  oIDateTimeProp As IDateTimeProp = oIProperty
Console.WriteLine(
         "Year={0}, Month={1}, Day={2}, Hour={3}, Minute={4}",
         oIDateTimeProp.Year,
         oIDateTimeProp.Month,
         oIDateTimeProp.Day,
         oIDateTimeProp.Hour,
         oIDateTimeProp.Minute)
```

Note that setting the string value on IProperty to a string representation of a date-time will result in the string being parsed and the date-time value will be set.

6.7.4. IBooleanProp Interface

For Boolean property types, the IBooleanProp interface is available…

Property	Description
IProperty As IProperty	A reference back to the IProperty interface for this property.
Value As Boolean	The Boolean value.

To use this interface for boolean properties, for example…

```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty("MyBoolProperty", True)

Dim oIBooleanProp As IBooleanProp = oIProperty
If (oIBooleanProp.Value) Then
```

```
   ‘ Do Something
End If
```

Note that setting the string value on IProperty to either “True” or “False” will result in the string being parsed and the Boolean value will be set.

6.8. Enumerated Properties

For numeric properties that have units of measure, the property type will be some class of UoMListType; e.g. PressureUoM. For all UoMListTypes, the IUoMListProp interface is defined…

Property	Description
EnumEnum As IObject	A reference to the enum value (schema).
GetUniqueName As String	???
IEnumHierarchy	A reference to the IEnumHierarchy interface. This is useful for multi-level enumerated lists. See the next section for more information on this.
IProperty As IProperty	A reference to the IProperty interface for this property.
Unparsed As String	If there was an error in parsing, this property will contain the portion of the string value not parsed.
ValueAsNumber As Boolean	If true, indicates that the string value is the number of the enum.
ValueAsText As Boolean	If true, indicates that the string value is the short text of the enum.
ValueAsUID As Boolean	If true, indicates that the string value contains the UID of the enum.

To use this interface for enumerated properties, for example…

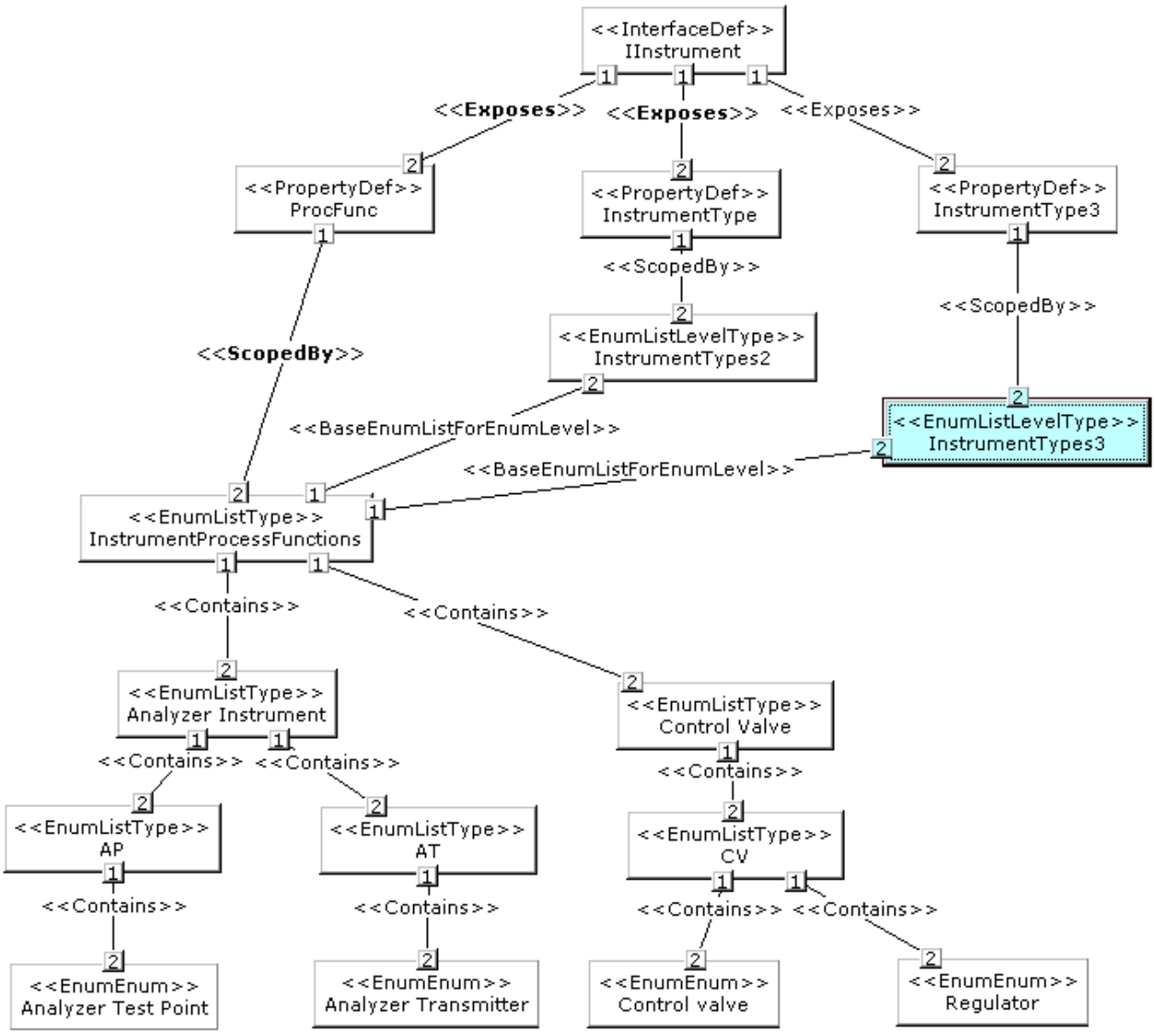
```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty(“MyEnumProperty”, True)

Dim oIEnumListProp As IEnumListProp = oIProperty

‘ Print out the short text and the UID of the enum value.
Console.WriteLine(
         “Enum value: Text={0}, UID={1}”,
         oIEnumListProp.EnumEnum.Name,
         oIEnumListProp.EnumEnum.UID)
```

Note that setting the string value on IProperty to a string having the short text value of (e.g. “Red”), the UID (e.g. “@EE427”) or the number (e.g. “#123”) an enum will result in the value being parsed into the component parts on IEnumListProp.

6.9. Hierarchical Enumerated Lists and Properties

Enumerated lists may contain other enumerated lists that go n-levels deep. These *hierarchical* lists allow a user to narrow down valid values for properties based on other property values. There is an object type called *EnumListLevelType* that enables this hierarchical structure.

If you review the IInstrument interface for PIDInstrument, you'll notice three of the exposed properties are related in that they are based on the same EnumListType (called *InstrumentProcessFunctions*). The UML diagram below demonstrates this:



The ProcFunc property is scoped by InstrumentProcessFunctions directly; thus its valid values might be "Analyzer Instrument" or "Control Valve." The InstrumentType property is scoped by an *EnumListLevelType* called *InstrumentTypes2*.

This property's values may come from the second level of the hierarchy that starts at the related base EnumListType: in our example, "AP", "AT", and "CV."

The InstrumentType3 property is scoped by an *EnumListLevelType* called *InstrumentTypes3*.

This property's values may come from the third level of the hierarchy that starts at the related base EnumListType: in our example, "Analyzer Test Point", "Analyzer Transmitter", "Control Valve", and "Regulator."

The hierarchy is useful in that the values for properties associated with the lower levels of the hierarchy are limited based on values at a higher level. For example, if ProcFunc = "Control Valve", then InstrumentType = "CV", and InstrumentType3 = "Regulator" or "Control valve."

There is functionality that allows you to set an nth-level property value, and all levels above that level will be set appropriately. The following example shows code that sets InstrumentType3, which in turn will automatically set InstrumentType2 and ProcFunc.

```
Dim oInstrument As IObject
Dim oClassDefComponent as IClassDefComponent
Dim oInterface As IInterface
Dim oProperty As IProperty
Dim oEnumHierarchy as IEnumHierarchy

'// Get object (logic not shown)
Set oInstrument = ...

'// Get IInstrument interface
Set oClassDefComponent = oInstrument
Set oInterface = oClassDefComponent.GetInterface("IInstrument", True)

'// Get any property on the interface (doesn't matter which level)
Set oProperty = oInterface.GetProperty("ProcFunc", True)

'// Set property to "Regulator" (EE426) at lowest level, which will set
'// InstrumentType2 = "CV" and ProcFunc = "Control Valve"
Set oEnumHierarchy = oProperty
OEnumHierarchy.SetHierarchyToEnumUID oInterface, "EE426"
```

6.10. UoM Properties

For numeric properties that have units of measure, the property type will be some class of UoMListType; e.g. PressureUoM. For all UoMListTypes, the IUoMListProp interface is defined…

Property	Description
CommaDelimitedValue As String	A string representation with all components of the UoM value delimited by commas.
ConditionIObj As IObject	A reference to the condition attached to this value.
DefaultSIUoM As IObject	A reference to the default SI UoM. This is the UoM enum in the schema; not the value in the default SI UoM.
DisplayUoMEnumIObj As IObject	A reference to the UoM enum (schema) to which this value is currently set.
EnglishTextValue As String	
IProperty As IProperty	A reference to the IProperty interface for this property.
Precision As Short	The number of decimal places to which this value is precise.
SIConditionIObj As IObject	The default SI condition.
SIValue As Double	The value in the default SI UoM.
TextValue As String	The string representation of this value.
Uncertainty As String	An optional uncertainty component for this value.
Unparsed As String	If the parser fails to completely parse the string value, this property will contain the balance of the string.
UomListTypeIObj As IObject	A reference to the UoMListType in the schema.
ValueInDisplayUoM	The real value in the current display UoM.

As Double

To use this interface for UoM properties, for example…

```
Dim oIProperty As IProperty =
oIMyInterface.GetProperty(“MyUoMProperty”, True)

Dim oIUoMListProp As IUoMListProp = oIProperty

‘ Print out the display value and uom
Console.WriteLine(
      “Display: Value={0}, UoM={1}”,
      oIUoMListProp.ValueInDisplayUoM.ToString(),
      oIUoMListProp.DisplayUoMEnumIObj.Name)

‘ Print out the SI value and uom
Console.WriteLine(
         “SI: Value={0}, UoM={1}”,
         oIUoMListProp.SIValue.ToString(),
         oIUoMListProp.DefaultSIUoM.Name)
```

Note that setting the string value on IProperty to a string having the value and UoM (e.g. “23.5psi”) will result in the string being parsed and the properties on IUoMListProp will all be set.

6.11. Lab Exercise

1. Using the code from the last lab, set the value of the entire instrument type hierarchy to “Relief Device”, “PSV I”, “Press/vac relief valve I”.
    a. Get the first instrument from the data container.
    b. Get a reference to the IInstrument interface.
    c. Get a reference to the ProcFunc property.
    d. Get a reference to the IEnumHierarchy for the ProcFunc property.
    e. Set the entire hierarchy using IEnumHierarchy to the value “Press/vac relief valve I” (you must use the UID for that enum).

You may use the Chapter 6 – Lab 3 folder as a starting point or continue to build upon Lab 2.

The completed lab exercise is available in the *Solution* project folder.

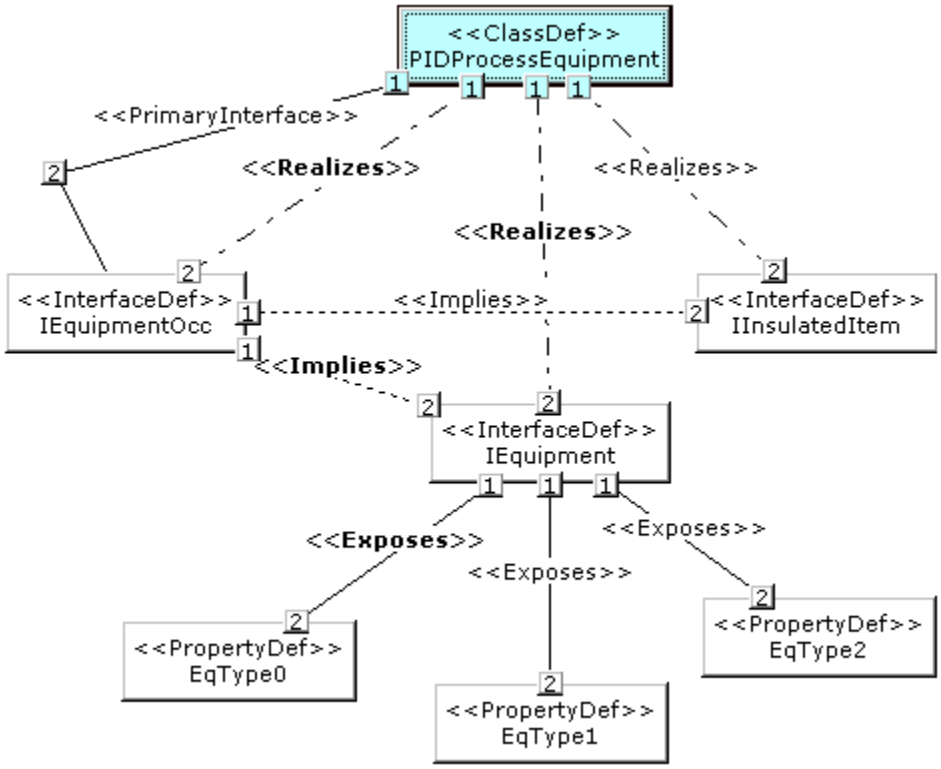
7. Creating Objects and Relationships

7.1. Creating Objects

To create a new instance of an object, get the class definition (*IClassDef*) for the object from the schema and use the *CreateInstance* method. For example, to create a new instance of "PIDProcessEquipment", use code similar to the following:

```
'// Example of creating a new piece of equipment
Dim oObject As IObject
Dim oClassDef As IClassDef
Dim oEquipment As IObject

'// Get the PIDProcessEquipment IObject from the Schema
Set oObject =
oSchemaContainer.IContainerQuery.GetIObjectForUID("PIDProcessEquipment")

'// Get the ClassDef object
Set oClassDef = oObject

'// Create an instance in the data container
Set oEquipment = oClassDef.CreateInstance(oDataContainer)
```

When you call CreateInstance, all *required* interfaces and properties are created for the object. Required interfaces and properties display as **bold** in the Schema Editor as in the following picture:



You can right-mouse click on the Relationship (*Realizes*, *Exposes*, etc.) to toggle its required property on or off. You may also select *Edit Relationship* from the popup menu that displays when you right-mouse click on a relationship. Then go to the *Properties* page and toggle the *Is Required?* field to the desired value (*True* or *False*).

CreateInstance adds the new object to the specified (data) container. However, this alone does not make it a valid object. The UID for the object, as well as the other required and known properties for the object, should be defined and set once the object has been created.

For all objects, IObject and any Primary Interface Definition (and all of its implied interfaces) are required interfaces.

7.2. Creating Relationships

To create a new Relationship object, the approach is similar to creating an instance of any other object; use the *CreateInstance* method on the *IRelDef* interface:

```
'// Create the Relationship object
Dim oObject as IObject
Dim oRel As IRel
Dim oRelDef as IRelDef

Set oRelDef =
oSchemaContainer.IContainerQuery.GetIObjectForUID("MyRelationDef")
Set oObject = oRelDef.CreateInstance(oDataContainer)
```

Alternatively, you may create relationships with the *CreateRelationship* method on the *IRelHelper* object. This function simplifies the creation somewhat because it handles setting the UID, UID1, UID2, and DefUID properties. For example, to complete the definition of the relationship shown previously, the following additional code is required:

```
'// Set the UID for the relationship
Set oObject.UID = "MyObj1RelatesToMyObj2"

'// Get the IRel object interface
Set oRel = oObject

'// Set the two ends of the relation
Set oRel.UID1IObj = oMyObj1
Set oRel.UID2IObj = oMyObj2
```

The corresponding logic using the CreateRelationship helper function is:

```
'// Using the Relationship Helper
Dim oRel As IRel
Dim oRelHelper As IRelHelper

'// Create the helper
Set oRelHelper = New Helper

'// Create the relationship
Set oRel = oRelHelper.CreateRelationship(oDataContainer,
"MyObj1RelatesToMyObj2", "MyObj1", "MyObj2", "MyRelationDef")
```

Note that the third and fourth arguments are variants and may be either the IObject for the objects involved or the UIDs of the two objects involved.

7.3. Lab Exercise

1. Write a VB program that creates an instance of PIDInstrument, an instance of PIDInstrumentLoop, and an InstrumentLoopAssembly relationship between them.

    a. Use IContainerQuery.GetIObjectForUID to get a reference to the PIDInstrument ClassDef.

    b. Call IClassDef.CreateInstance to create the new instrument.

c. Create a new UID and set the name to PT-14403. Set the instrument type hierarchy to ‘Pressure Transmitter’.

d. Make sure you instance the InterfaceDef ILoopMember or you will get a validation error for the InstrumentLoopAssembly rel.

e. Use GetIObejctforUID to get a reference to the PIDInstrumentLoop ClassDef and create and instance of that.

f. Create a new UID for the loop and set the name to P-14403.

g. Using IRelHelper.CreateRel, create an instance of the ‘InstrumentLoopAssembly’ RelDef.

You may use the Chapter 7 – Lab 1 folder as a starting point.

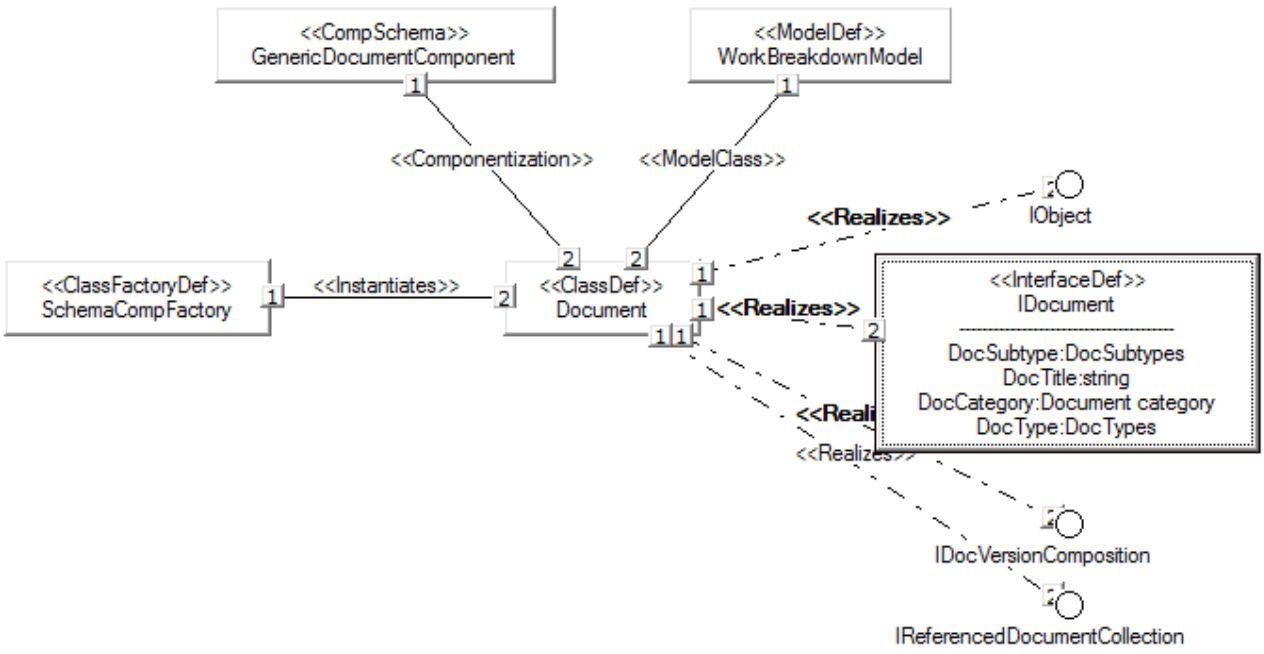
The completed lab exercise is available in the *Solution* project folder.

7.4. Creating Documents

On common task you will need to perform is to create instances of documents. The document object defines the meta data around the data you will be publishing or retrieving.

When you create a ComponentSchema for your application, you will need to create a document ClassDef as part of that ComponentSchema. However, you will never need to create an instance of it yourself. When you must create an instance of a document, it will be of the generic document ClassDef 'Document'. This is because it usually occurs at a time before the SmartPlant Client software knows what the ComponentSchema is going to be. Therefore, it loads the GenericDocument ComponentSchema only. It will later transform the document from ClassDef Document to your specific document ClassDef.

The class diagram for Document is as follows…



In addition to the UID and Name properties on IObject, you must also set all the properties on IDocument. DocTitle is self-explanatory. DocCategory, DocType, and DocSubtype are all part of the document type hierarchy. These properties are scoped by the 'Document category' EnumListType.

Also notice the IDocVersionComposition InterfaceDef. This is needed for the relationship to the version objects. Make sure you instance that InterfaceDef.

7.5. Lab Exercise

1. Write a VB program that creates an instance of the ClassDef Document.
    a. Use IContainerQuery.GetIObjectForUID to get a reference to the Document ClassDef.
    b. Call IClassDef.CreateInstance to create the new document.
    c. Create a new UID and set the name. Set the document type hierarchy to 'Control System Configuration'.

You may use the Chapter 7 – Lab 2 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

S E C T I O N 2

Adapter Authoring

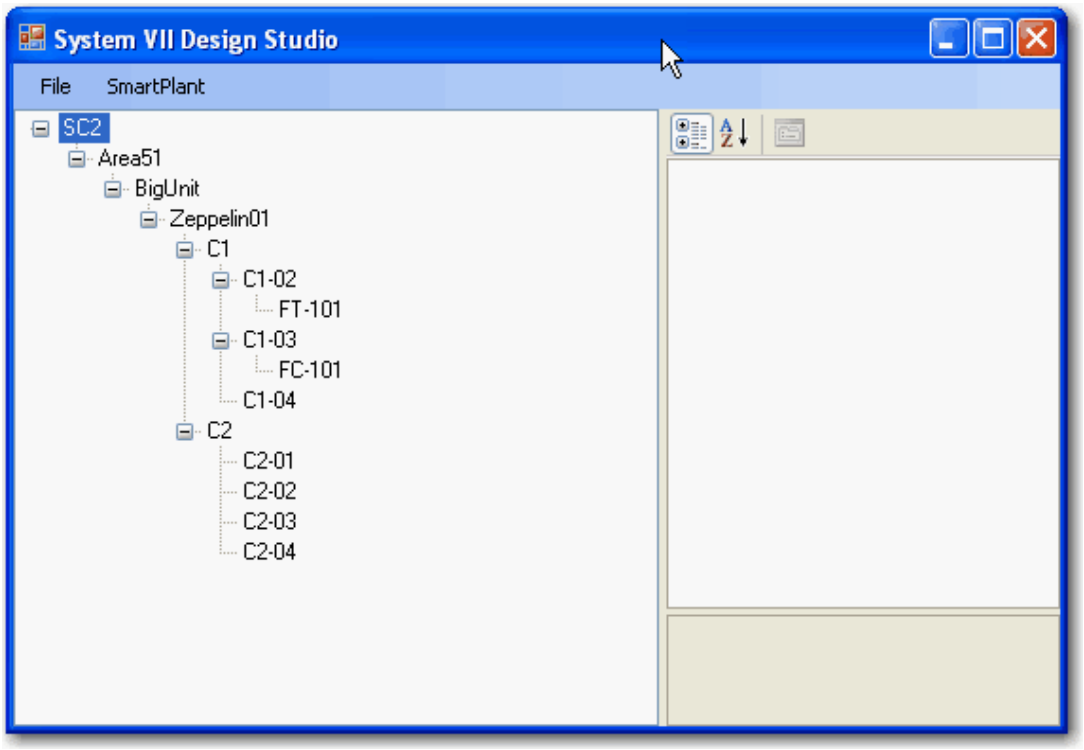
8. Adapter Overview

For this section of the course, we're going to be working with a fictitious application. We'll suppose that we are software developers for a vendor of digital control systems. Our application is the control system design software for the latest generation of control systems called 'System VII'.

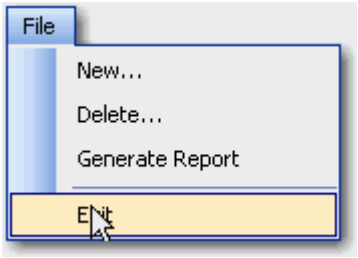
While this example application is fictitious, it does highlight a real-world example in which Intergraph has been very active. We have partnered with most of the largest DCS vendors in the world to create SmartPlant Enterprise integrations with their design software. This is a very important workflow supporting collaboration between the instrument engineering discipline, using SmartPlant Instrumentation, and the control system design discipline. The schema we'll be using in this example is the same schema used in the actual DCS integrations.

8.1. User Interface

Our example application is called ‘System VII Design Studio’. The user interface is very simple displaying a treeview, loaded with the hierarchy of data and a property grid from which you can view object properties.



A File menu is exposed providing access to a few application-specific commands…

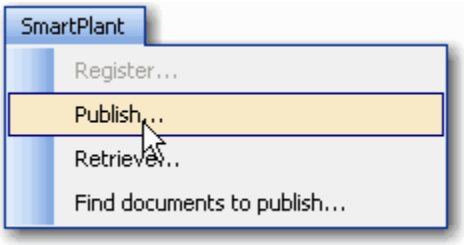


New command – This will create any type of business object; the type depending on the selected object. You select the parent object and choose ‘New’ to create a child object.

Delete command – This will delete any selected business object.

Generate Report command – This application will generated a simple report of all IO tags within a given controller. A controller or one of it's children must be selected to run this command. This report illustrates the view file that will be published with any controller.

A SmartPlant menu is exposed providing access to the most command SmartPlant Enterprise commands. As part of this course, you will be completing the code to implement these commands…



Register command – Once implemented, used to register this application with SmartPlant Foundation. Registration is between a plant in the application and a plant in SPF, so a plant or one of it's children must be selected in order to execute this command.

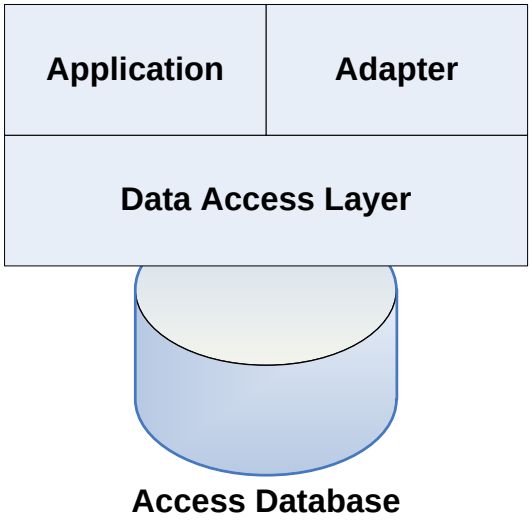
Publish command – Once implemented, used to publish a controller configuration document to SmartPlant Foundation. The scope of a publish is a controller, so a controller or one of it's children must be selected to execute this command.

Retrieve command – Once implemented, used to retrieve documents from SmartPlant Foundation.

Find documents to publish – Once implemented, used to discover documents that need to be published or documents that have been deleted because the controller was deleted.

8.2. Architecture

The application is composed of three VB.NET projects. The application, the adapter, and the data access layer. The data store is implemented as a Microsoft Access database.



The application is implemented as a standard Windows Forms application. Almost all of the code is implemented in the main form (MainF.cls).

The adapter will be implemented as a VB.NET Class Library. The adapter itself will be a single class within this library.

The data access layer is also implemented as a VB.NET Class Library. A class has been implemented for every business object understood by this application. All business objects inherit from a class called Sys7Entity. This super class defines the following properties…

CacheState – This property tracks states of New, Changed, Deleted, and Unchanged. This property determines how the object is written to the database; Insert, Update, Delete, or not at all.

Id – A GUID is assigned to every object when it is created.

Name & Description – All objects are given standard data properties of name and description.

ParentId – The Id of the parent is stored in every child object.

SameAsId – To be explained in a later chapter.

The following methods are defined at the super class…

GetMapClass – Virtual function that returns the name of the tool map class.

GetValue & SetValue – Virtual functions that will get or set a value based on the map property name.

ReadById, ReadByName, ReadBySameAs – These functions will read in an object based on Id, Name, or SameAsId. Each of these functions will be used later in the course to implement a correlation algorithm.

Read – Two functions that actually encapsulate the database read.

ResolveParentId – A virtual function that will resolve a parent relationship from external UID to internal.

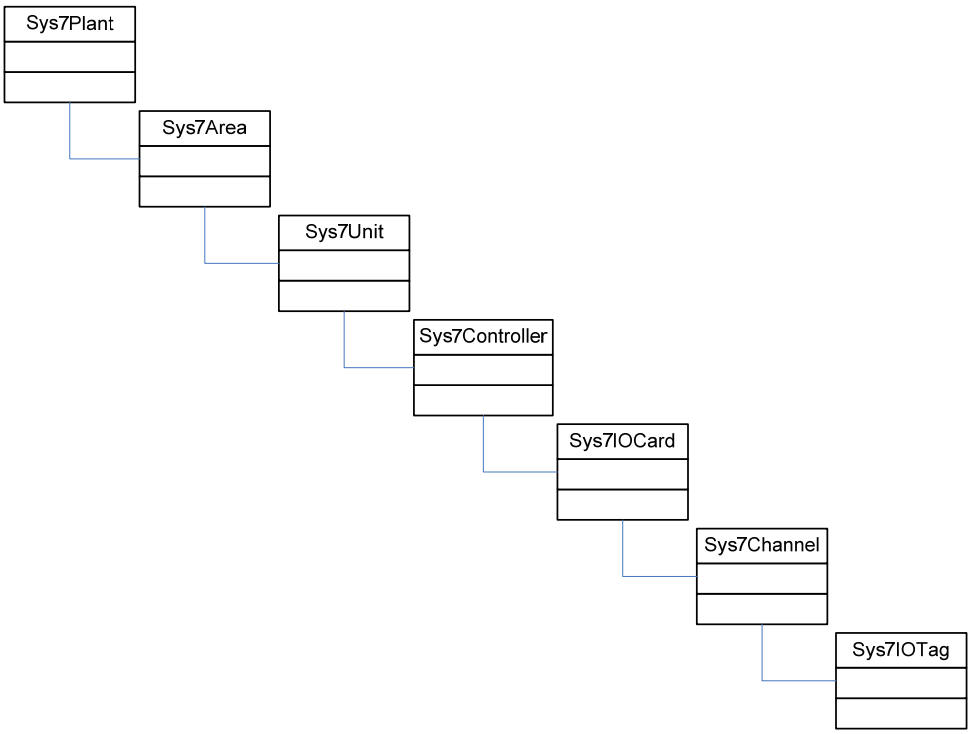
ReadChildren – A virtual function that will return a collection of the children of this object.

Write – Simply call write on any business object to correct persist it.

Insert, Update, & Delete – Worker functions called by Write. Don't call these directly.

SetChanged, SetDeleted – Used to change the CacheState value. Whenever you change a property on a business object, the SetChanged function should get called.

Business objects inheriting from Sys7Entity are organized into a hierarchy as follows…

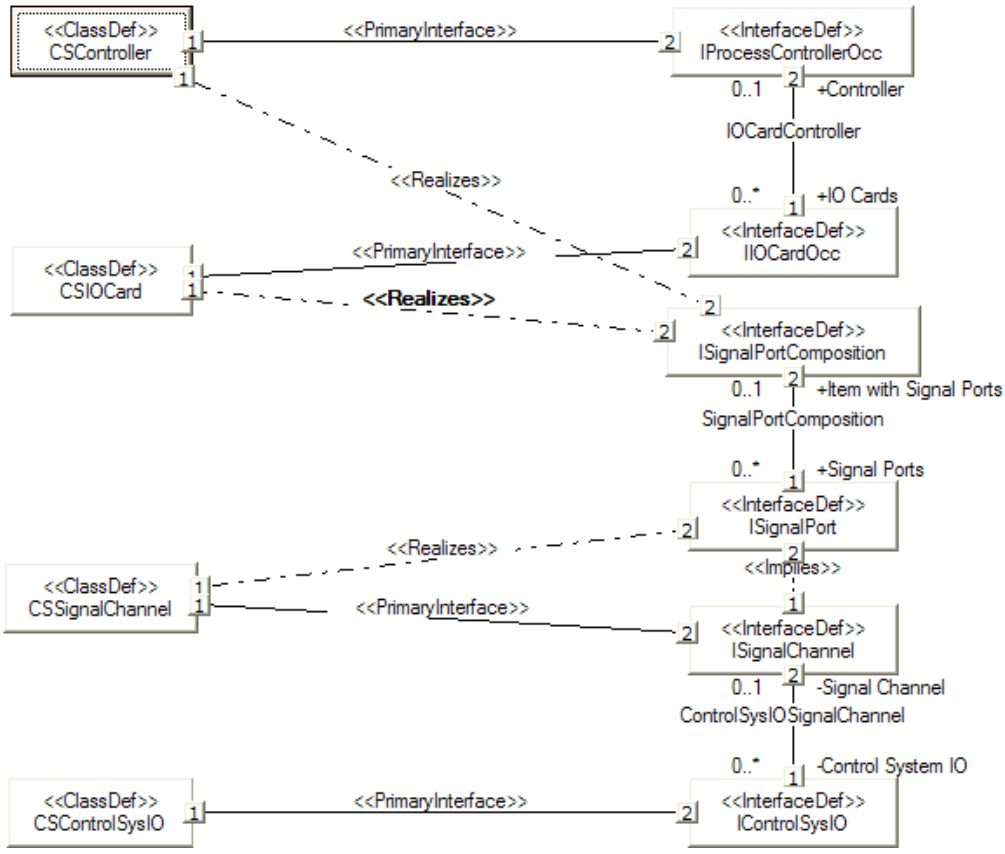


Each of the business object classes implement some specific properties.

A class named Sys7ClassFactory will create any business object based on the tool map class name. This will be used in the final chapter to implement mapping.

8.3. Schema

The following diagram shows a small portion of the control system schema…



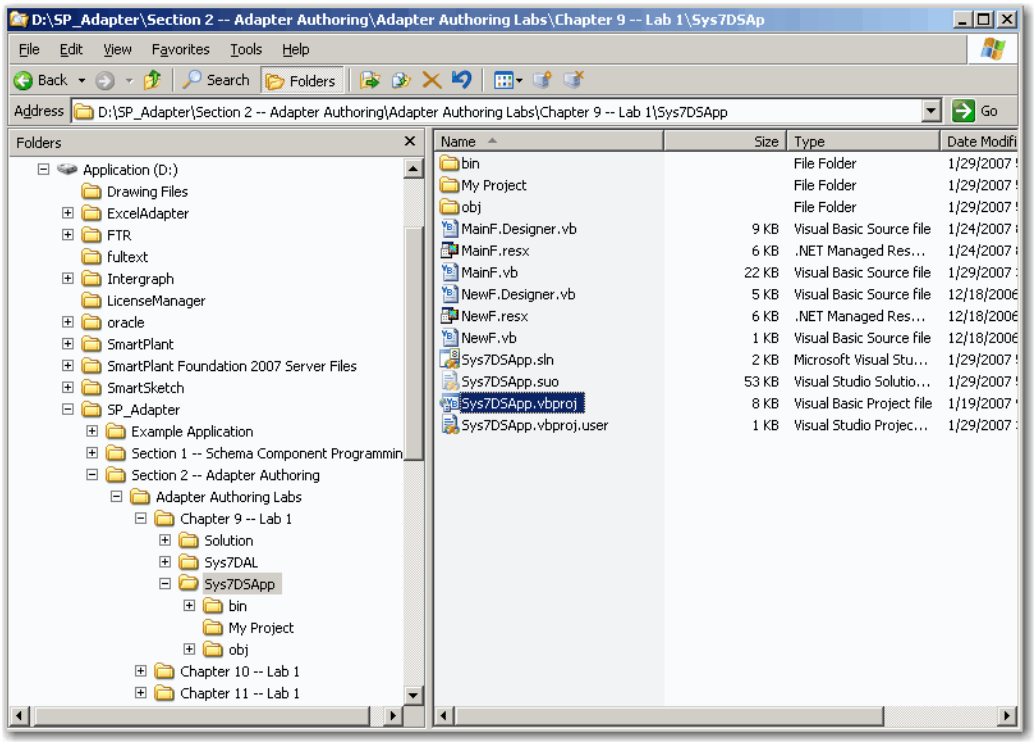
At the top of the control system hierarchy is the controller. A controller is essentially a computer that is executing software to automate control of the plant. The controller does it's work through IO cards, which are the translators from the various inputs & outputs in the field to the all-digital controller. An IO card generally has many channels, each of which are connected to some device in the field. The field device is represented logically by a control system tag. This is how the control system addresses the device.

The control system system is connected to the plant breakdown structure through the PBSItemCollection rel; CSController realizes IPBSItem. The PBS schema could actually be anything, but we will assume plant, area, unit for this course.

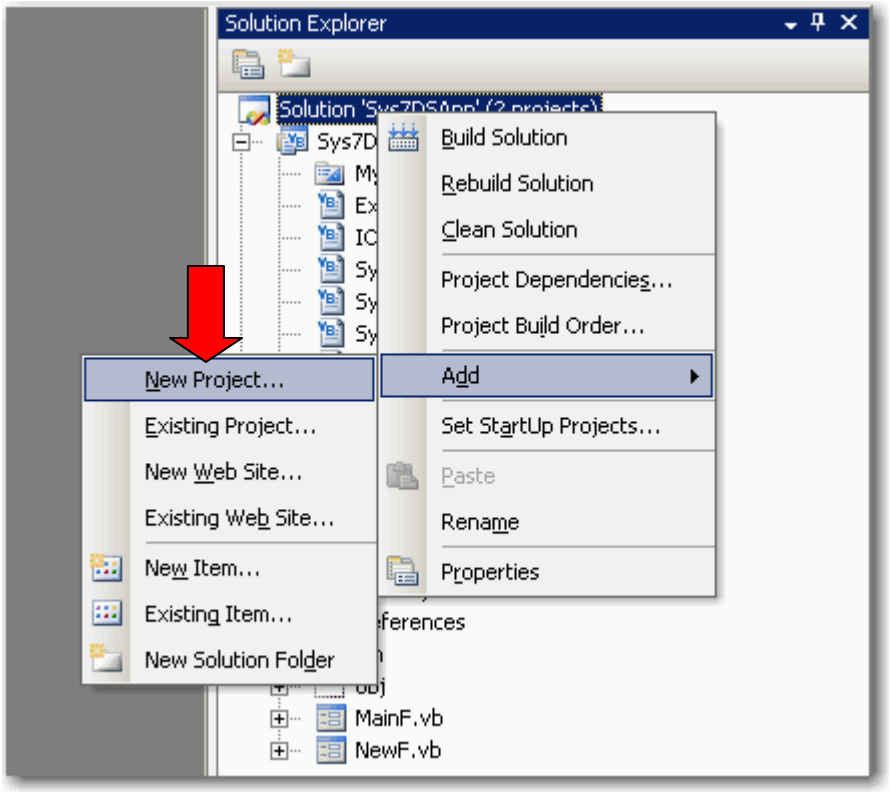
9. Creating the Adapter Shell

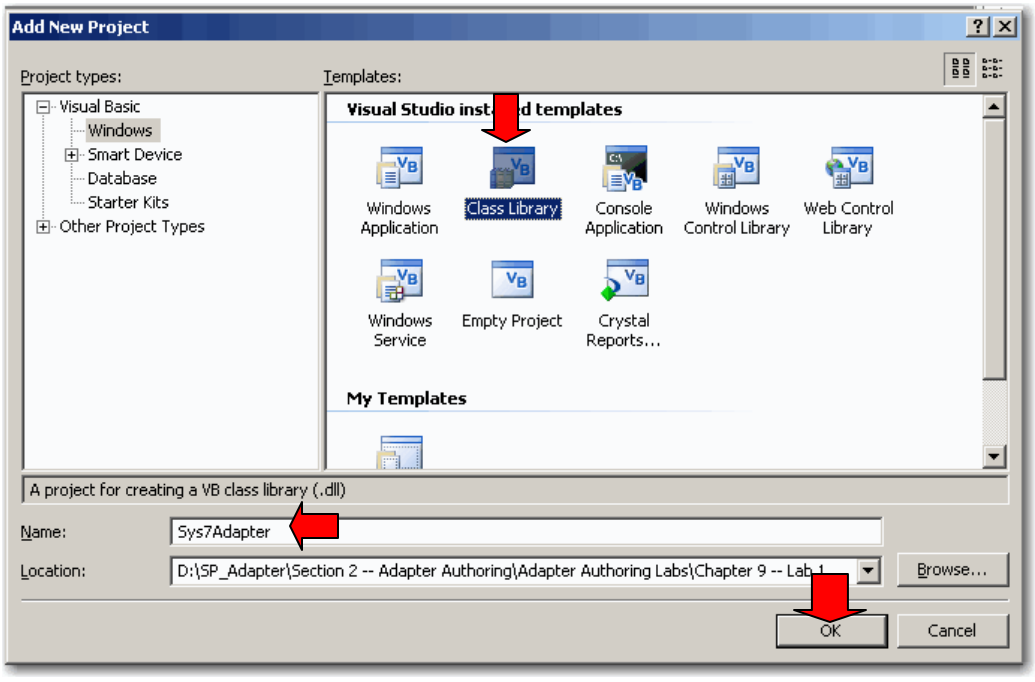
The first step in creating a TEF Adapter is to create the basic framework for the adapter. We refer to this framework as the *Adapter Shell*. This framework is simply a VB.NET Class DLL project that implements the *IEFAdapter* interface.

To start creating your adapter shell, start Visual Basic .NET and open the Sys7DSAPP application code (Sys7DSApp.vbproj).

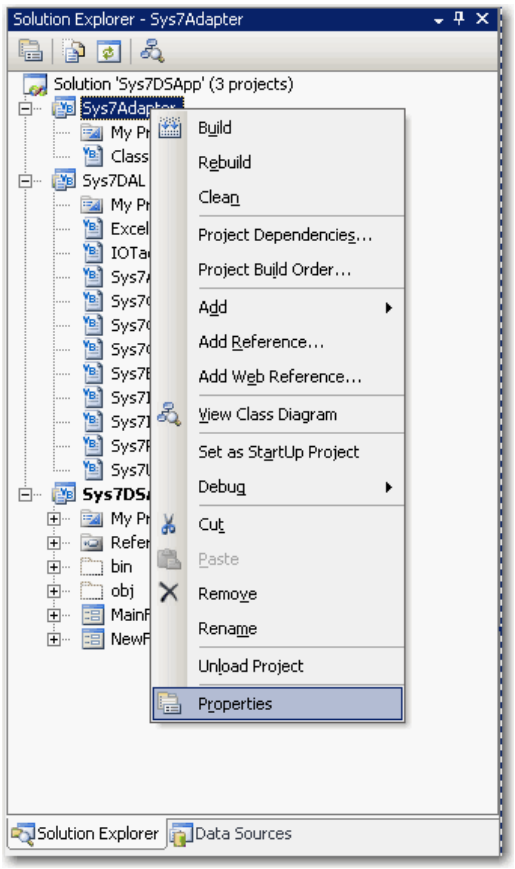


Once open RHMB the Solution and Add a new Project

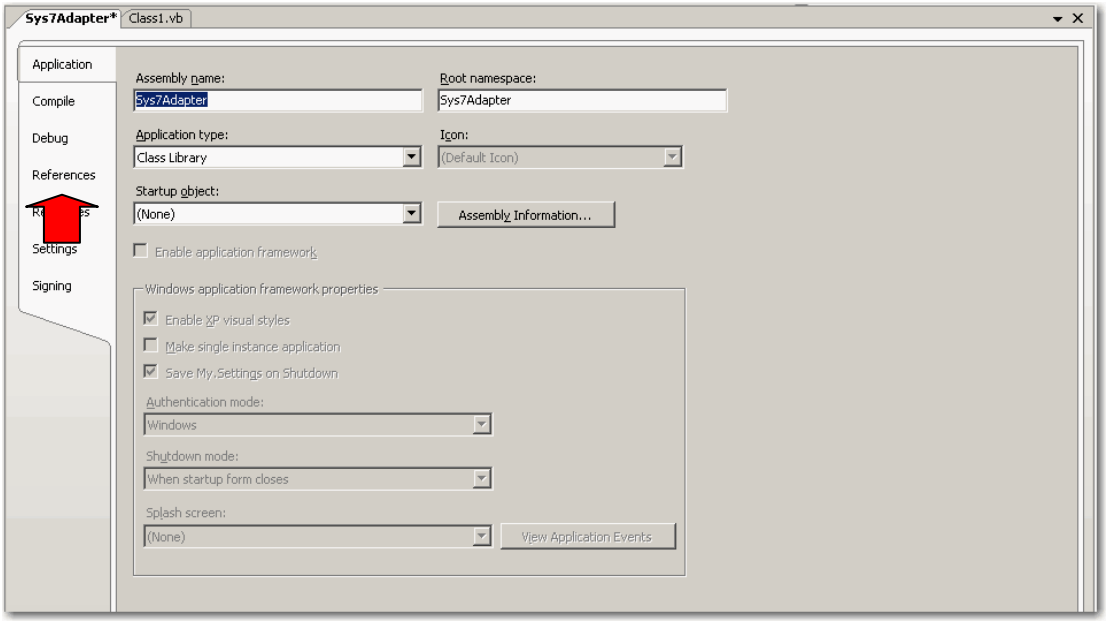


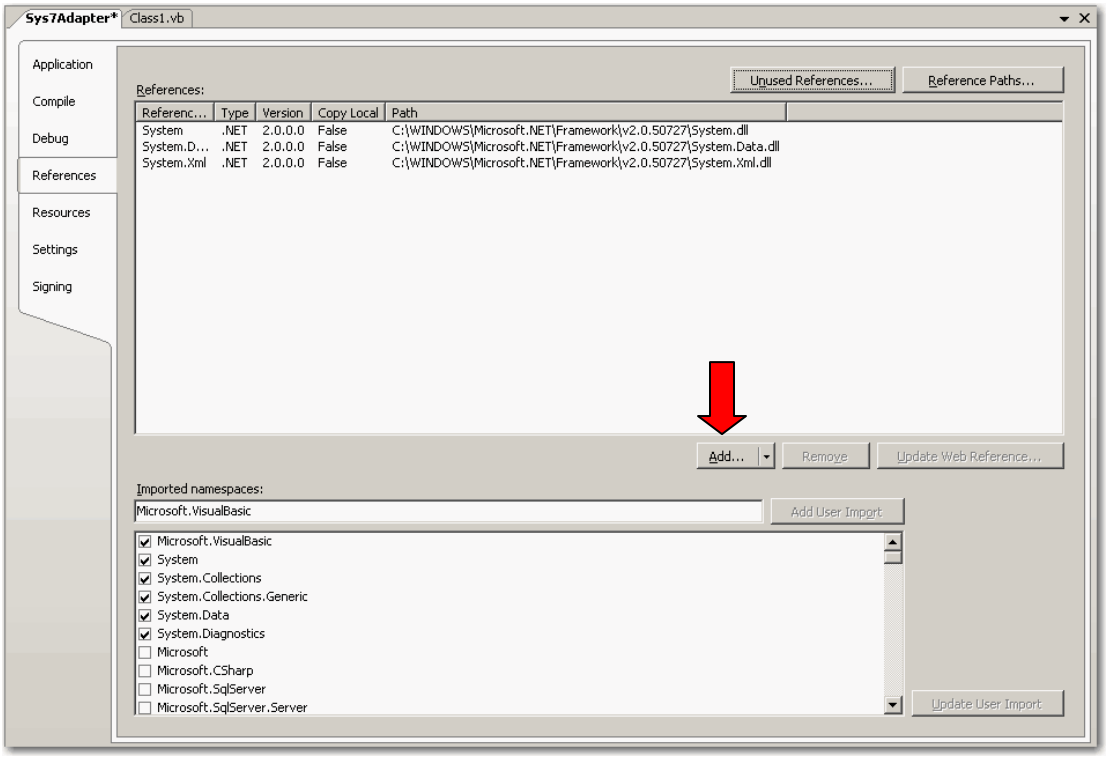


Under the new **Sys7Adapter** RHMB and select properties.



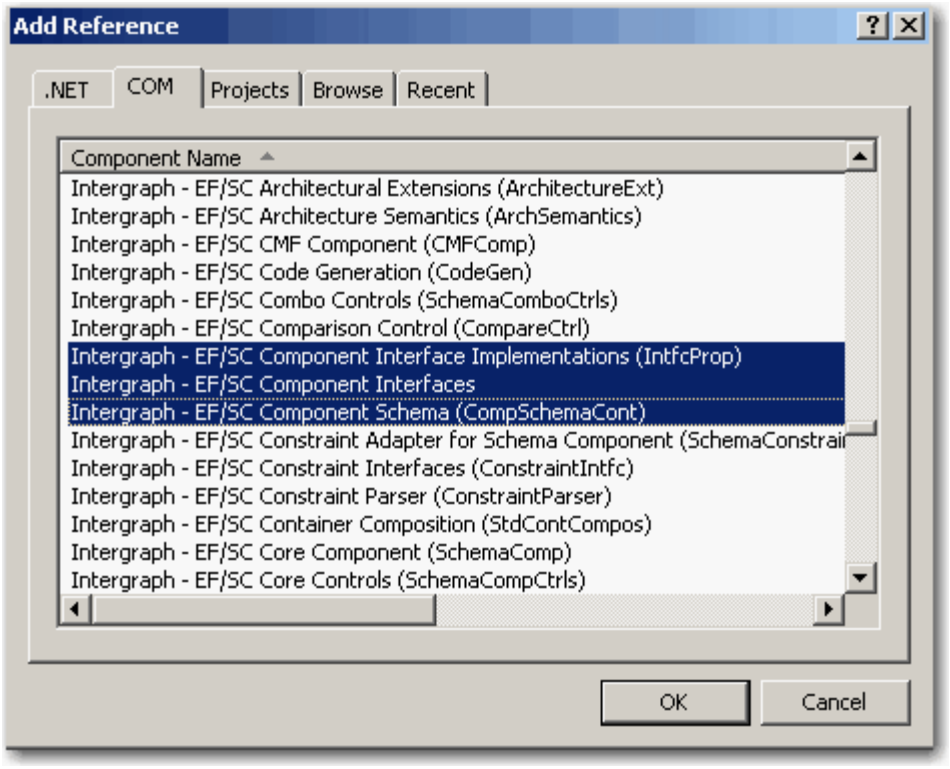
The property dialog appears



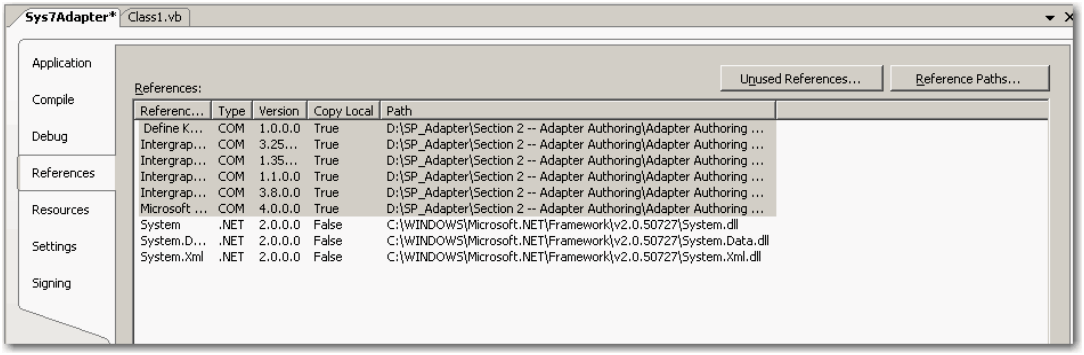


Add references to the following type libraries:

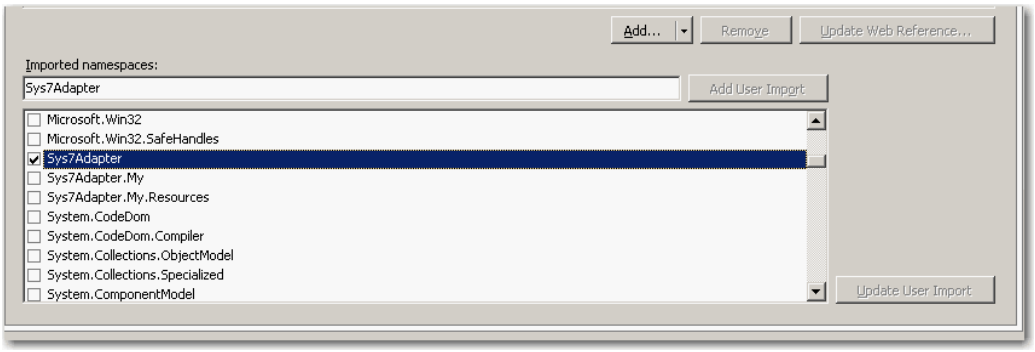
- *Intergraph – EF/SC Component Interface Implementations (IntfcProp)*
- *Intergraph – EF/SC Component Interfaces*
- *Intergraph – EF/SC Component Schema (CompSchemaCont)*

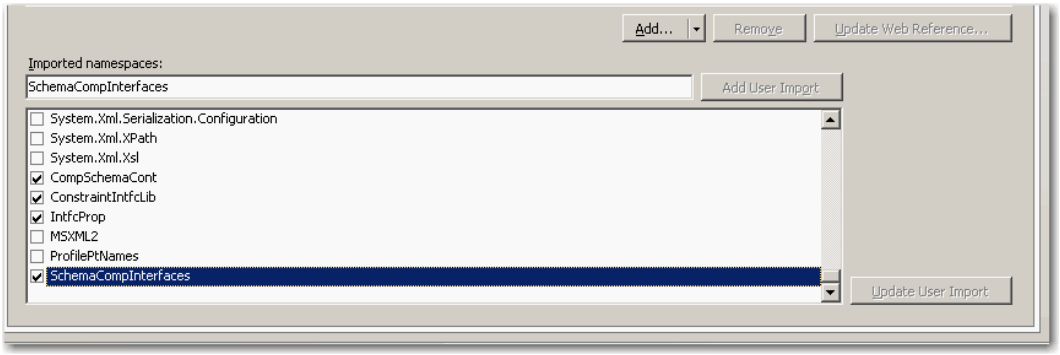


The references will be added to the master list

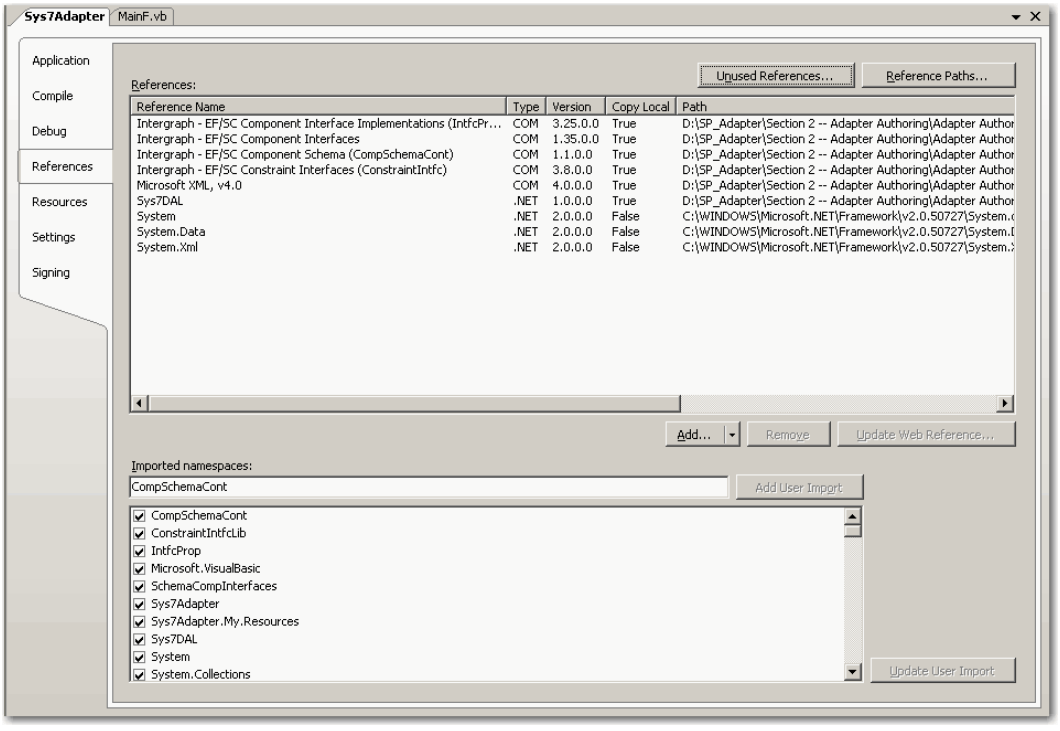


Next add at tablespace's to the project, Add **Sys7Adapter, CompSchemaCont, ContrantIntfclib, IntfcProp, SchemaCompInterfaces and Sys7DAL**.





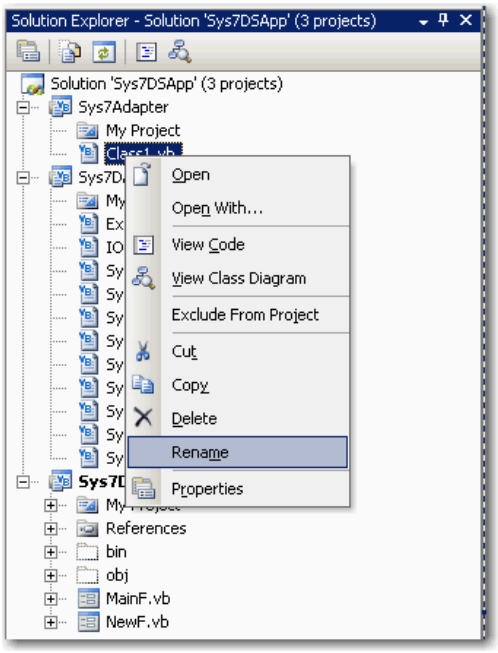
The final reference dialog setup should look like this:

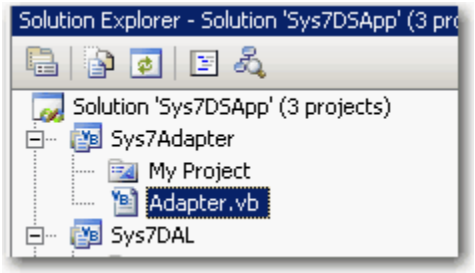


Next, you should give your project and class meaningful names. These names form the *Programmatic Identifier*, or *ProgID*, for your adapter.

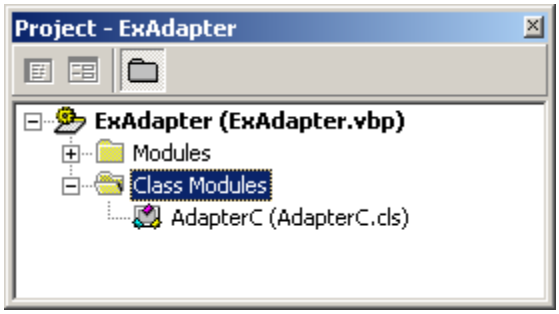
Note: *The ProgID is always ProjectName.ClassName.*

Rename Class1 to Adapter under the Sys7Adapter project.





The ProgID is used when you register your adapter with *SmartPlant* and also to instantiate your adapter whenever it is needed. The example adapter shell for this course has a ProgID of *Sys7Adapter.Adapter* as shown in the project view window.



This is a good time to save your project before we begin adding code.

9.1. IEFAdapter

The *IEFAdapter* interface is a required interface that all TEF Adapters must implement. The interface has the following properties and methods:

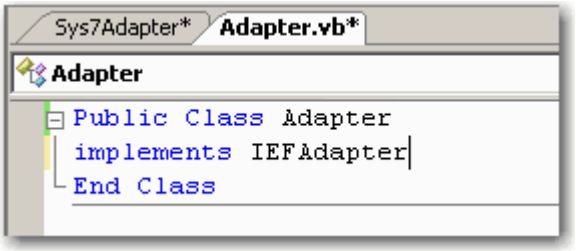
Property	Description
CurrentSPFProjectStatus As SPFProjectStatus	Sets the current project status.
MapSchemaIContainer As IContainer	Sets the map schema container.
MessageIContainer As IContainer	Sets the message container.
Signature As String	Gets the signature of the tool.
SPFPlant As String	Sets the SmartPlant Foundation plant.
SPFProjectName As String	Sets the SmartPlant Foundation project name.
SPFProjectUID As String	Sets the project unique identifier.
SPFURL As String	Sets the URL for the SmartPlant Foundation server.
ToolID As String	Sets the tool ID.
ToolParameters As IEFToolParameters	Sets the collection of tool parameters.

Method	Description
DocumentExistsInTool(oDocumentIObj As IObject) As Boolean	Checks whether the specified document exists in the tool database.
FindDocsToPublish(oDocsToPublishIContainer As IContainer, oNewDocsToPublishIContainer As IContainer)	Finds documents to publish.

Method	Description
GetCompSchemaUID(sDocTypeUID As String) As String	Returns the component schema UID for a given document type.
GetDocClassDefUID(sDocTypeUID As String) As String	Returns the class definition UID of document for a given document type.
GetDocumentList(sDocTypeUID As String, oDocIContainer As IContainer)	Returns a collection of document IObjects for a given document type.
GetGraphicFile(oDocumentIObj As IObject, sGraphicFileName As String)	Gets the associated graphics file for a publishing document.
GetMapSchemaFile(sDocTypeUID As String) As String	Returns the mapping schema file name (full path).
GetObjectStatusInTool(sObjUID As String, sClassUID As String) As ObjectStatusInTool	Returns the object status in the design tool.
GetPublishableDocTypes() As IEFDocTypes	Returns a collection of document types that a tool can publish.
GetRelationshipStatusInTool(sRelUID As String, sClassUID As String) As ObjectStatusInTool	Returns the relationship status in the design tool.
GetRetrievableDocTypes() As IEFDocTypes	Returns a collection of document types that a tool can retrieve.
GetRevision(oDocumentIObj As IObject) As String	Returns the revision value for a given document.
PublishConfirm(oDocumentIObj As IObject)	Persists the fact that the document has been published.

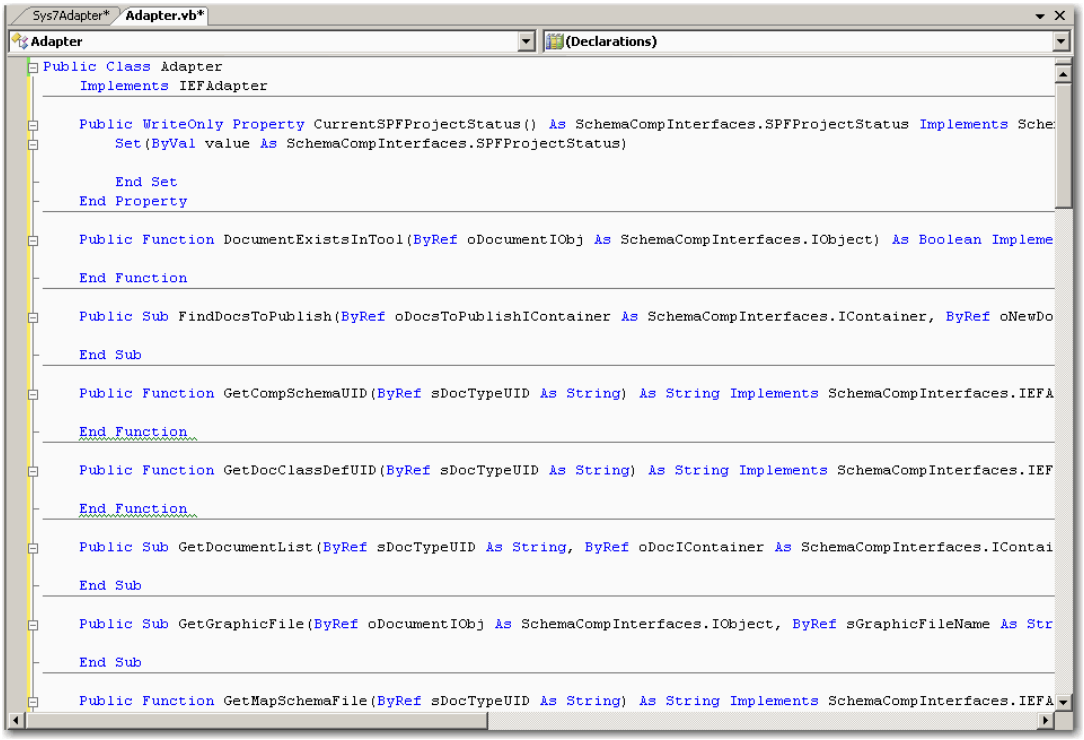
Method	Description
PublishDocument(oDocumentIObj As IObject, oDocContIContainer As IContainer, oDocMetaIContainer As IContainer, bViewFileRequired As Boolean, [oMapSchContIContainer As IContainer])	Adds design tool data objects into document data container and the meta container.
RetrieveDocument(oDocumentIObj As IObject, oDocContIContainer As IContainer, oDocTombstonesIContainer As IContainer, oDocMetadataIContainer As IContainer, [oMapSchContIContainer As IContainer])	Imports the data objects in containers into the design tool.
SupportsFeature(Feature As ToolFeatures, [sDocTypeUID As String]) As Boolean	Checks whether the specified feature is supported in the tool.
Terminate()	Releases all of the resources the adapter has obtained.

Visual Basic allows you to implement interfaces using the *Implements* keyword (for further information on implementing interfaces, see the Visual Basic Help file topic *Using Interfaces with Visual Basic*). In your Visual Basic class module declarations section, add the line *Implements IEFAdapter*.

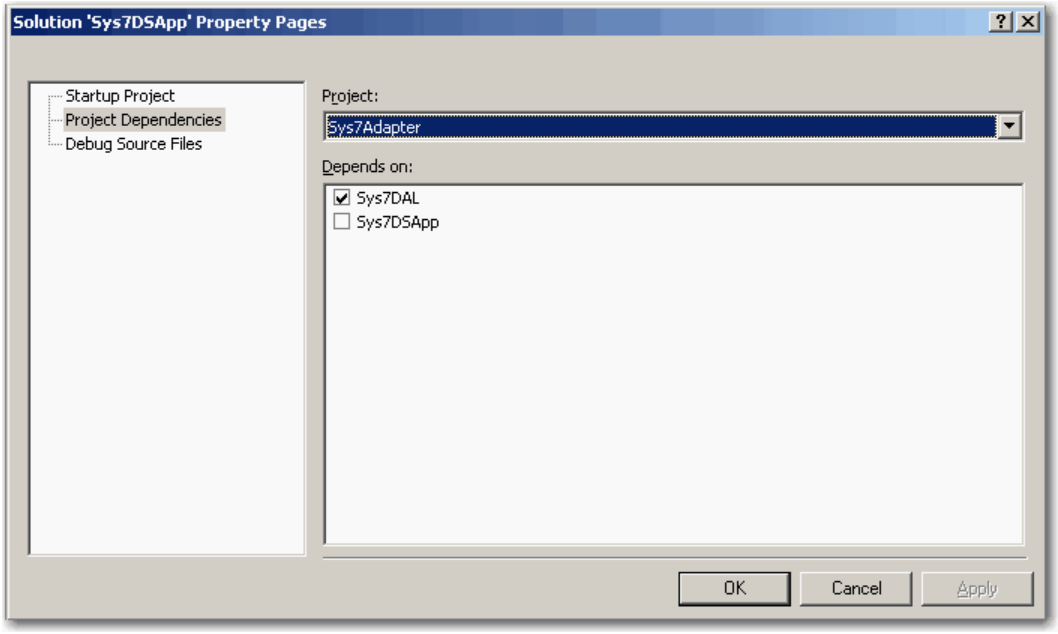


9.2. Implementing the IEFAdapter Interface

Once the implements line is added all the Properties and functions will be added to the class.



Under the solution RHMB and select properties. Under the **Project Dependences**, Select the **Sys7Adapter** and select **Sys7DAL** as a dependent project.



9.3. Lab Exercise

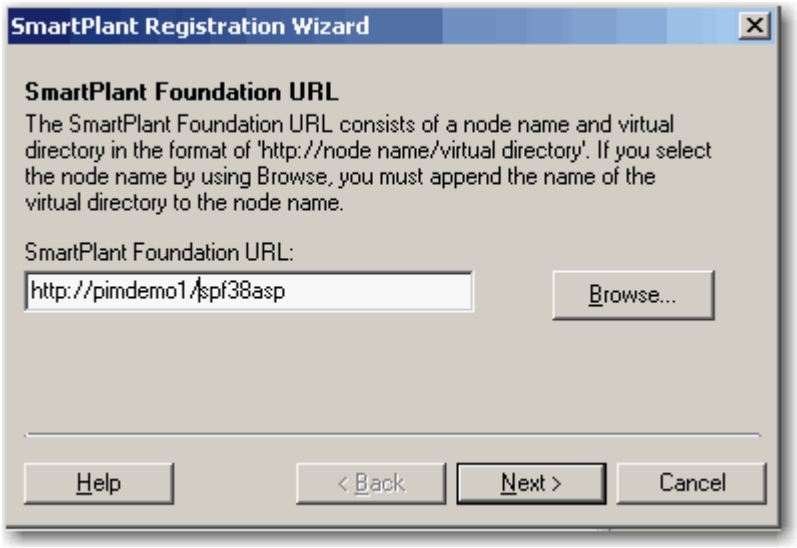
Perform all of the steps in this chapter to create an Adapter Shell and get it to compile in debug mode.

You may use the Chapter 9 – Lab 1 folder as a starting point.

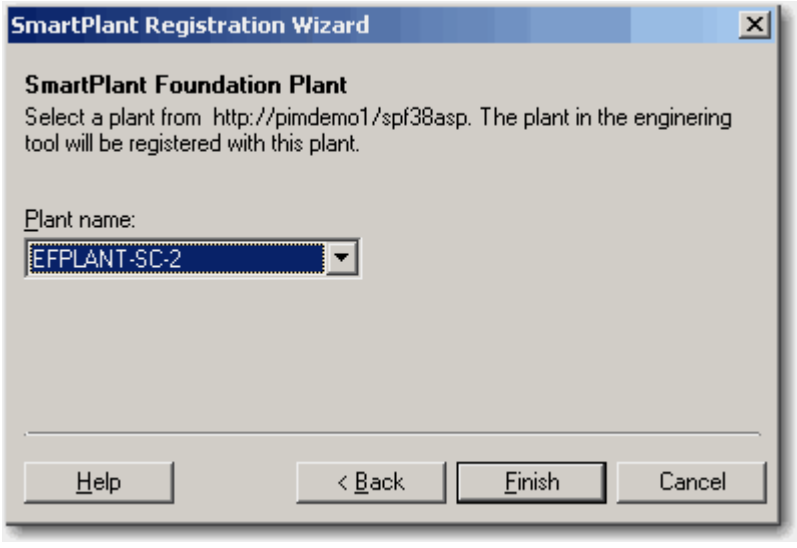
The completed lab exercise is available in the *Solution* project folder.

10. Registering with SmartPlant

You must *register* your application and adapter before it can connect and integrate with *SmartPlant*. To register with SmartPlant, your application (not the adapter) must create an instance of the SmartPlant Client and call the *Register* method. This method returns values that you must store for your application to use whenever it communicates with *SmartPlant*. The following are the dialogs you'll see for registering a Plant.



After you enter the URL for your SPF server, select Next. The Plant dialog will appear.



Click Finish to complete the process.

The SmartPlant Client implements an interface called IEFCommonUIApplication through which your application will call properties and methods.

10.1. IEFCommonUIApplication

Property	Description
Connected As Boolean	Boolean indicating connection status to SmartPlant Foundation.
CurrentSPFProjectStatus As SPFProjectStatus	Current SmartPlant Foundation project status. (Project/as-built mode only.)
EFAdapter As IEFAdapter	Pointer to IEFAdapter.
LastErrorMessage As String	Most of the methods will return a long value indicating success or failure. If failure, this property will contain the error message generated by the last method call.
SPFPlant As String	SmartPlant Foundation plant. (Project/as-built mode only.)
SPFProjectName As String	SmartPlant Foundation project name. (Project/as-built mode only.)
SPFProjectUID As String	SmartPlant Foundation project unique ID. (Project/as-built mode only.)
SPFURL As String	SmartPlant Foundation URL.
ToolParameters As IEFToolParameters	Pointer to design tool parameters.

Method	Description
AutoRetrievalOptions() As Long	Displays the Automatic Retrieval Options dialog box. (Auto retrieve is an option feature of the SmartPlant Client. For this to work, the application must expose functionality through a web service. This is outside the scope of this course.

Method	Description
ClaimObjects(oIContainer As IContainer, [sDocUID As String]) As Long	Claims objects after GetClaimContainer is called. (Project/as-built mode only.)
ClaimObjectsInXML(sDocUID As String, oClaims As DOMDocument) As Long	Claims objects using XML format. (Project/as-built mode only.)
Connect(sSPFURL As String, sSPFPlant As String, sSPFProjectUID As String, sSPFProjectName As String, sToolID As String, sToolAdapterProgID As String, oToolParameters As IEFToolParameters, oToolDocs As IEFToolDocs, [bFileMode As Boolean]) As Long	Most methods (Publish, Retrieve, FindDocsToPublish) require the application to connect to SmartPlant first. (The exception is Register.)
FindDocsToPublish() As Long	This method starts the Find Documents to Publish process. This will be covered in detail in Chapter 18.
GetClaimContainer(oIContainer As IContainer, [sCompSchemaUID As String], [sDocTypeUID As String]) As Long	Creates the schema component objects required for claim operation. (Project/as-built mode only.)
GetComponentSchema(sCompSchemaUID As String, sCompSchemaFile As String) As Long	Returns a specified component schema file.
GetDocumentListContainerForAsBuiltPublish (sProjectUID As String, oDocsForAsBuiltPublishIContainer As IContainer, oSelectedDocIContainer As IContainer) As Long	(Project/as-built mode only.)
GetDocumentListContainerForPublish(oDocIContainer As IContainer) As Long	Returns a data container scoped by the Generic Document component schema allowing the application to specify active or selected documents to be published prior to displaying the SmartPlant Common UI Publish dialog.

Method	Description
GetUnclaimContainer(oIContainer As IContainer, [sCompSchemaUID As String], [sDocTypeUID As String]) As Long	Creates the schema component objects required for unclaim operation. (Project/as-built mode only.)
PublishAllAsBuiltDocuments(sProjectUID As String, bProgressBar As Boolean, sLogFile As String) As Long	Publishes as-built documents to a project. (Project/as-built mode only.)
PublishSpecifiedAsBuiltDocuments(sProjectUID As String, oDocIContainer As IContainer, bProgressBar As Boolean, sLogFile As String) As Long	Publishes the specified as-built documents to a project. (Project/as-built mode only.)
Register(sToolID As String, sToolDescription As String, sToolEFAdapterProgID As String, oToolParameters As IEFToolParameters, sSPFURL As String, sSPFPlant As String, sSignature As String, [oPBSIContainer As IContainer]) As Long	Starts the Register Wizard leading the user through the registration process.
ReleaseClaimContainer() As Long	Rolls back the claim operation initiated by GetClaimContainer method. (Project/as-built mode only.)
ReleasePublishDocumentListContainer() As Long	Releases the containers created in GetDocumentListContainerForPublish method.
ReleaseUnclaimContainer() As Long	Rolls back the unclaim operation initiated by GetUnclaimContainer method. (Project/as-built mode only.)

Method	Description
ShowEFCommonUI() As Long	Displays the SmartPlant Foundation Web Client within a dialog box. This provides access to two features unavailable otherwise: 1) Retrieving superceded versions of documents. 2) Comparing current application data with a previous publish. ***Note***: this will be replaced in a future version with specific methods to invoke these features.
ShowPublishDialog(oDocIContainer As IContainer) As Long	This method begins the Publish process resulting in the display of the SmartPlant Common UI Publish dialog.
ShowRetrieveDialog([oDocTypes As IEFDocTypes]) As Long	This method begins the Retrieve process resulting in the display of the SmartPlant Common UI Retrieve dialog.
UnclaimObjects(oIContainer As IContainer) As Long	Unclaims objects after GetUnclaimContainer is called. (Project/as-built mode only.)

There are many properties and methods on the IEFCommonUIApplication interface that we discuss later; however, for the time being, we are going to focus on the *Register* method.

In order to register your application with *SmartPlant*, your adapter must be implemented and compiled (which we accomplished in the previous chapter) and you must know the ProgID (*ProjectName.ClassName*) of your adapter.

Register method arguments

In this chapter, we're going to learn how to implement the Register command within our application. The bulk of the work is handled by the call to the Register method on the SmartPlant client. The follow table explains the paramters to that method.

Argument	Type	Required	Description
sToolID	String	Y	A short name for SmartPlant to use for your application.
sToolDescription	String	Y	A longer description for SmartPlant to use for your application.
sToolEFAdapterProgID	String	Y	The COM ProgID of your adapter.
oToolParameters	IEFToolParameters	Y	The tool parameters collection for your application. **Note** The collection may be empty but must be present.
sSPFURL	String	Y	The returned URL string for the SmartPlant Foundation server.
sSPFPlant	String	Y	The returned plant string for the SmartPlant Foundation plant.
sSignature	String	Y	The returned signature string for your application/plant combination.
oPBSIContainer	IContainer	N	An optional variable. If provided, then the return value is a container for the plant area system.

10.2. IEFToolParameters & IEFToolParameter

For most of the SmartPlant operations (including register), a tool parameters argument is provided. This argument allows the application to pass any required data to the adapter; through the SmartPlant Client. There are two interfaces involved in this transfer: IEFToolParamters, which is a collection; and IEFToolParameter, which is simply a name-value pair.

The methods and properties on the IEFToolParameters interface are as follows…

Property	**Description**
Count As Integer	The number of tool parameters in this collection.
Item(Object) As IEFToolParameter	Returns the tool parameter from the collection identified by the key value passed in the variant argument.

Method	**Description**
Add(Name As String, Value As Object, Key As String) As IEFToolParameter	Creates a new tool parameter and adds it to the collection. The Key value can be used later to access the objects via the Item property.
Remove(IndexKey As Object)	Removes the tool parameter identified by the index key argument.

Let's look at a Visual Basic code example for registering an application. It is customary to add a 'SmartPlant' menu to the application's main menu bar containing a 'Register' command. The implementation of this command should go through these basic steps…

- Create an instance of the SmartPlant Client
- Create an instance of the ToolParameters class.
- Call the Register method on the SmartPlant Client.
- Persist the registration info.

Here is the actual code for our example application...

```
        Try
            ' Create an instance of the SmartPlant Client
            Dim commonUIApplication As IEFCommonUIApplication
            commonUIApplication = CreateObject("EFCommonUI.Application")
            If commonUIApplication Is Nothing Then
                Throw New Exception("Error starting SPF Common UI.")
            End If

            ' Create an instance of ToolParameters
            Dim toolParameters As IEFToolParameters
            toolParameters = New EFToolParameters()

            ' Declare some local variables for the returned registration
information
            Dim spfUrl As String = ""
            Dim spfPlant As String = ""
            Dim spfSignature As String = ""
            Dim result As Integer = 0

            ' Call the Register method
            result = commonUIApplication.Register("Sys7DS", "System VII Design
Studio", "Sys7Adapter.Adapter", toolParameters, spfUrl, spfPlant,
spfSignature)

            ' Check for success or failure
            If result < 0 Then
                Throw New Exception("Error during Register operation. [" &
commonUIApplication.LastErrorMessage & "]")
            Else
                MessageBox.Show("Register completed succussfully", "Register
Completed", MessageBoxButtons.OK)
            End If

            ' Persist the returned information against our internal plant
            Dim plant As Sys7Plant = GetSelectedPlant()
            plant.SpfUrl = spfUrl
            plant.SpfPlant = spfPlant
            plant.SpfSignature = spfSignature
            plant.Write(_db)

        Catch exp As Exception
            MessageBox.Show("An error occurred during the register process. "
& exp.Message, "Publish", MessageBoxButtons.OK)
        End Try
```

When you call the register method, a series of dialog boxes appear. These dialog boxes are the Login dialog box, the SmartPlant Foundation URL dialog box, and the SmartPlant Foundation Plant dialog box. You must provide valid entries on these dialog boxes for the method to succeed.

The register method populates the sUrl, sPlant, and sSignature strings and returns them to you. Since an application can only be registered once for a given plant in the SmartPlant Foundation database, it is important that you save this information; associated with the application's internal plant. When the user is working within an internal plant in the application, it should enable/disable the SmartPlant menu commands depending on whether that plant is registered.

In the example application, these values are properties on the Plant table. The application has a method InitializeSmartPlant menu that can be called to enable/disable the SmartPlant menu commands based on the currently selected plant.

10.3. Lab Exercise

Add code to the sample application (RegisterMenu_Click) provided in the class examples folder to register the sample application with SmartPlant.

You may use the Chapter 10 – Lab 1 folder as a starting point.

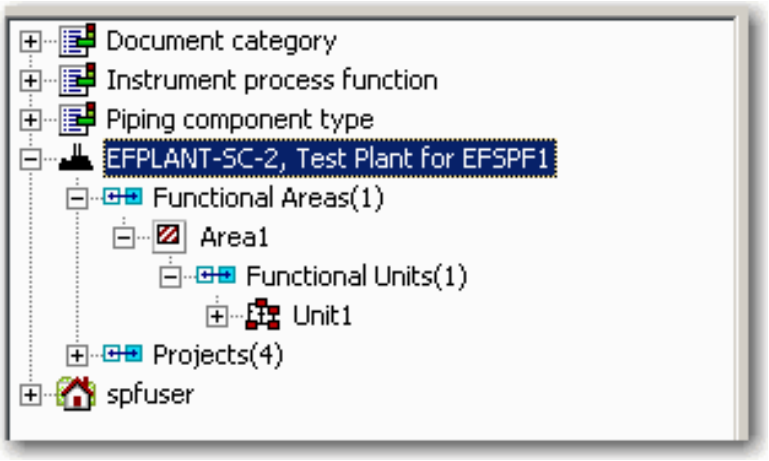
The completed lab exercise is available in the *Solution* project folder.

Once you have completed the lab, go ahead and register the application with SmartPlant by running the application.

1) Select *SC2*.
2) Select the *SmartPlant->Register* pulldown menu entry.
3) Key in the following *SmartPlant Foundation URL*: *http://pimdemo1/spf38asp* and then select *Next*.
4) Select *EFPLANT-SC-2* from the *Plant name:* combobox and then select *Finish*.

11. Retrieving Plant Breakdown Structure

Almost all plant design applications organize the data into some hierarchy within the plant. SmartPlant commonly refers to this as the Plant Breakdown Structure although there may be many names for it. To integrate with SmartPlant, all applications are asked to synchronize their PBS with SmartPlant Foundation. That way, when items are related into a particular location within the PBS, a retrieving app also has that exact location.



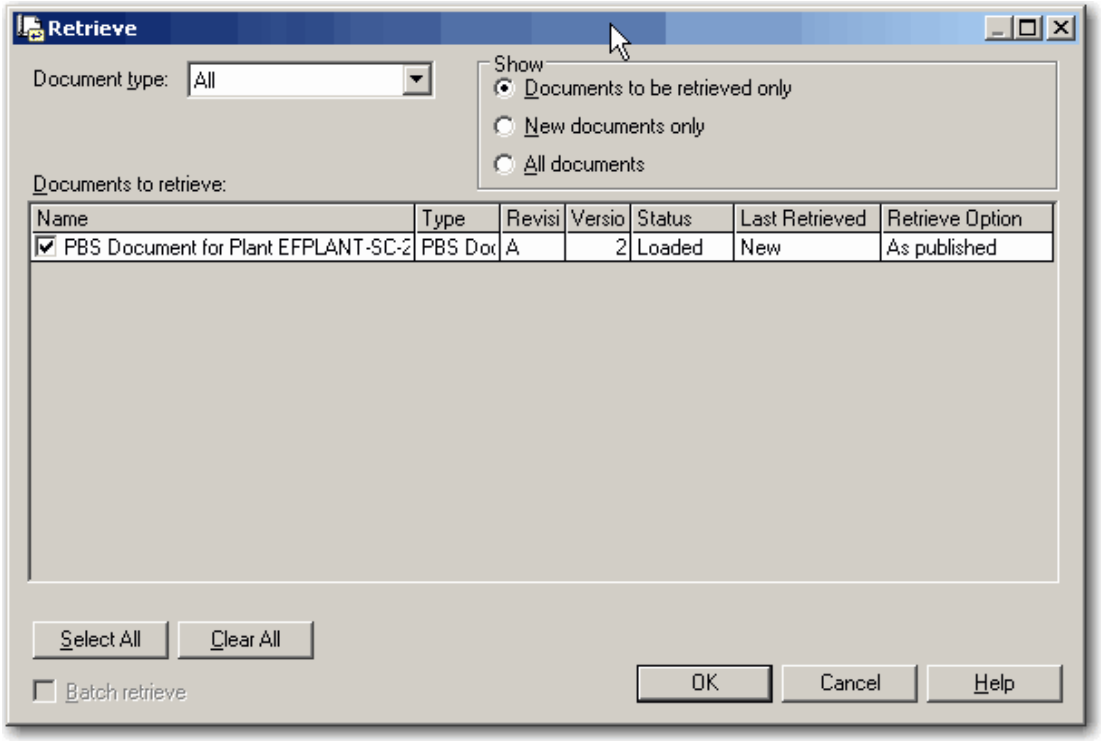
The schema for the PBS is variable because each customer may have a slightly or significantly different structure. Since even the schema may be different, an application must be prepared to retrieve PBS generically. However, in this example application, we are going to assume a PBS of Plant->Area->Unit to save time.

The PBS structure is created in SmartPlant Foundation and is published by an administrator using a method on the Plant object. This results in a PBS Document which will appear automatically in all application's Retrieve dialog the next time it is invoked. The PBS may also be created in parallel in the application so the application must be prepared to correlate PBS items when they already exist and create them when they do not.

11.1. Retrieve

SmartPlant integration is based on a publish/retrieve paradigm. Each authoring tool has a data store for its working data. When the author of a particular document or container of data decides that it is of sufficient quality to release it to others, he publishes it. Likewise, when someone needs to get data on which he will base his design, he retrieves it.

The GUI for retrieve is supplied by the SmartPlant Client. It is called the EF Common UI. The retrieve dialog is as follows…



Notice that there are three options under ‘Show’:

- Documents to be retrieved only – This option shows you only new revisions of documents you’ve (or this application) already retrieved. Documents that you have retrieved are considered to be part of your design basis of which SPF keeps track.
- New documents only – This option shows you new documents (you have not retrieved) of the type that your application can retrieve.
- All documents – All documents (new documents, new revisions and previously retrieved revisions) having document types that your application can retrieve.

11.2. Correlation

For now, we are going to use the term ‘Correlation’ to describe a process by which a retrieving application will decide whether an incoming object already exists in it’s database (and thus should result in an update to that object) or does not exists (and thus should result in a create). The algorithm each application is expected to implement for this process is four sequential steps...

1. Honor incoming correlations – if another application says it’s object is correlated to our, then honor that statement. Were going to ignore this step for now and come back to it in Chapter 17 – Same As.
2. If we have previously correlated an incoming object with an existing object, then this is an update.
3. Else, attempt to correlate the incoming object to an existing object by some set of property values (most commonly name). If this is successful, persist the correlation for later use.
4. Else, consider this a new objects to be created in the application database. Once created, however, it is correlated so the application should persist that for later use.

Any given object in SmartPlant may well be published by multiple applications. Take a control valve for instance. These may be published from Aspen Zyqad, SmartPlant P&ID, SmartPlant Instrumentation, SmartPlant Foundation (through the instrument process datasheet), and SmartPlant 3D. Because an object may come from multiple sources, each application is expected to many correlations to each object. This allows flexibility in the workflows that can be employed.

Note: As a simplifying constraint, our example application will persist only a single correlation against each of it’s objects. This is implemented as a property on each object class called SameAsId. When an object is correlated, the incoming UID is persisted to this property along with the internal identifier.

11.3. Code To Support PBS Retrieve

The changes we need to make to our example application and adapter to support retrieve of the PBS are as follows…

- Implement the menu handler in the application for the Retrieve command on the SmartPlant menu.
- Tell SmartPlant that we support retrieving the PBS document type by implementing the GetRetrievableDocTypes and GetDocClassDefUID methods in the adapter.
- Implement the ToolParameters method on the adapter to support the passing of information from the application.
- Implement the RetrieveDocument method on the adapter to consume the PBS data.

11.3.1. Implementing the Retrieve Menu Handler

The application code for the retrieve menu handler needs to follow these basic steps…

- Create an instance of the SmartPlant Client.
- Create and initialize the tool parameters.
- Create an instance of our tool docs class.
- Connect to SPF.
- Initiate the retrieve process.

All the work of consuming the data resulting from the retrieve will be the responsibility of the adapter.

To create an instance of the SmartPlant Client, we use CreateObject along with the COM ProgID as in the previous chapter…

```
            Dim commonUIApplication As IEFCommonUIApplication
            commonUIApplication =
CreateObject("EFCommonUI.Application")
            If commonUIApplication Is Nothing Then Throw New
Exception("Error starting SPF Common UI.")
```

Next, we need to create an instance of the tool parameters class and initialize it with some information our adapter will need. An application may use this name-value collection any way it chooses. For our example application, we're going to pass over the database connection and the current plant.

```
            Dim toolParameters As IEFToolParameters
            toolParameters = New EFToolParameters()
```

```
            If toolParameters Is Nothing Then Throw New
   Exception("Error instancing tool parameters.")
   tool parameters
```

Notice that adding items to the collection seems to require naming them twice. The first parameter is the name and the third parameter is a key value. If you will access the array by index, you will not need the key. We're going to ask for our values by the key so that we're less dependant upon position.

```
            toolParameters.Add("Database", _db, "Database")
            toolParameters.Add("Plant", plant, "Plant")
```

The method to connect to SPF requires an instance of our tool docs class. This class is only used during publish when the user clicks the 'Engineering Tool' button; it will not be used during retrieve. So, we don't have to add any implementation to that class, just make sure it implements the interface with stub methods. Here, we create an instance of that class. (We're creating it here using the CreateObject method, but we could simply use the New keyword as well.

```
            Dim toolDocs As IEFToolDocs
            toolDocs = CreateObject("Sys7Adapter.ToolDocs")
            If toolDocs Is Nothing Then Throw New Exception("Error
   instancing tool docs.")
```

Next, we need to connect to SPF. There is no difference between calling this method for register, retrieve, publish, or any of the other APIs.

```
            Dim result As Integer =
   commonUIApplication.Connect(plant.SpfUrl, plant.SpfPlant, "", "",
   "Sys7DS", "Sys7Adapter.Adapter", toolParameters, toolDocs)
            If result < 0 Then Throw New Exception("Error connecting
   to SPF. [" & commonUIApplication.LastErrorMessage & "]")
```

Now, to start the retrieve process, we simply need to call ShowRetrieveDialog. The SmartPlant Client will take over and lead the user through the retrieve process. Our adapter will be invoked during the process to consume the data for each document selected for retrieve.

```
            result = commonUIApplication.ShowRetrieveDialog()
            If result < 0 Then Throw New Exception("Error during
   Retrieve. [" & commonUIApplication.LastErrorMessage & "]")
```

The complete menu handler is as follows…

```
        Try
            ' Need the registration information out of the selected plant
            Dim plant As Sys7Plant = GetSelectedPlant()
            If plant Is Nothing Then Throw New Exception("Error getting
selected plant.")

            ' Create an instance of the SmartPlant Common UI
            Dim commonUIApplication As IEFCommonUIApplication
            commonUIApplication = CreateObject("EFCommonUI.Application")
```

```
            If commonUIApplication Is Nothing Then Throw New
Exception("Error starting SPF Common UI.")

            ' Set up the tool paramters
            Dim toolParameters As IEFToolParameters
            toolParameters = New EFToolParameters()
            If toolParameters Is Nothing Then Throw New Exception("Error
instancing tool parameters.")

            ' Send needed information over to the adapter through the
tool parameters
            toolParameters.Add("Database", _db, "Database")
            toolParameters.Add("Plant", plant, "Plant")

            ' Create an instance of the tool docs (even though we won't
be using it right now
            Dim toolDocs As IEFToolDocs
            toolDocs = CreateObject("Sys7Adapter.ToolDocs")
            If toolDocs Is Nothing Then Throw New Exception("Error
instancing tool docs.")

            ' Connect to SPF
            Dim result As Integer =
commonUIApplication.Connect(plant.SpfUrl, plant.SpfPlant, "", "",
"Sys7DS", "Sys7Adapter.Adapter", toolParameters, toolDocs)
            If result < 0 Then Throw New Exception("Error connecting to
SPF. [" & commonUIApplication.LastErrorMessage & "]")

            ' Start the publish process. The SmartPlant client takes over
from here and will make use of the adapter
            ' during the process.
            result = commonUIApplication.ShowRetrieveDialog()
            If result < 0 Then Throw New Exception("Error during
Retrieve. [" & commonUIApplication.LastErrorMessage & "]")

            ' Reinitialize the tree to display the new data
            InitializeTree()

        Catch exp As Exception
            MessageBox.Show("An error occurred during the Retrieve
process. " & exp.Message, "Retrieve", MessageBoxButtons.OK)
        End Try
```

11.3.2. Implementing GetRetrievableDocTypes and GetDocClassUID

Once the retrieve process has been started, the SmartPlant Client is going to ask the adapter for some information regarding it's retrieve capabilities. It needs to know what kinds of documents the adapter can retrieve so that it can display only those in the GUI. It will call the GetRetrievableDocTypes method on the adapter and expect it to return a collection of document types in an instance of the class EFDocTypes.

IEFDocTypes

Property	Description
Count As Long	The number of items in the collection
Item As IEFDocType	The item in the collection at the specified index or key

Method	Description
Add (sDocTypeUID As String)	Creates and adds an IEFDocType object to the collection of IEFDocTypes for the specified document type UID (string).
Remove (VKey As Variant)	Removes the specified item from the collection.

To implement this method in our adapter, we simply need to create an instance of EFDocTypes and then add in each of the document type UIDs that we can retrieve. For now, we know we need to retrieve the PBS Document type. Let's also go ahead and add in the other document type we'll eventually retrieve; 'SPI DCS configuration document'.

Note: Since this requires the document type UID, you will probably need to use the Schema Editor to get that information. If you search the EnumListTypes for 'doc*', it will return you an enum list named 'Document category'. If you view this list, you can see the entire hierarchy and view the correct UIDs.

Note: You will need a reference to 'EF Common UI Data Classes' to create an instance of EFDocTypes.

The code for our implementation is as follows…

```
        Dim docTypes As IEFDocTypes = New EFDocTypes
        If docTypes Is Nothing Then Throw New Exception("Error
creating retrievable document types.")
        ' Add the PBS Document
        docTypes.Add("{E4528C43-83A9-11D6-BD6F-00104BCC2B69}")
        ' Add SPI DCS configuration document
        docTypes.Add("{F84347CF-A57A-4CBE-894A-2F5B2ED1EE56}")
        Return docTypes
```

The SmartPlant Client is also going to ask for help on the document ClassDef associated with each document type. It will call the GetDocClassDefUID for each document type and the adapter is expected to return the correct ClassDef. The code for our implementation is as follows…

```
GetDocClassDefUID = Nothing
Select Case sDocTypeUID
    Case "{E4528C43-83A9-11D6-BD6F-00104BCC2B69}"
        ' PBS document
        GetDocClassDefUID = "PBSDocument"
    Case "{F84347CF-A57A-4CBE-894A-2F5B2ED1EE56}"
        ' SPI DCS Configuration
        GetDocClassDefUID = "CSDocument"
End Select
```

11.3.3. Implement ToolParameters Property

To pass the tool parameters collection to the adapter from the application, the SmartPlant Client uses the ToolParameters property Set on the adapter. There is nothing required by SmartPlant during this call; it is completely up to the adapter implementation what it needs to do here. For this example implementation, the database connection and the current plant have been passed in from the application. Our adapter will simply pull those from the collection using the key values and tuck them away into member variables for later use. Our code is as follows…

```
m_toolParameters = value
m_db = m_toolParameters.Item("Database").vValue
m_plant = m_toolParameters.Item("Plant").vValue
```

11.3.4. Implementing Signature

This property is called by the SmartPlant Client during the Connect method to ask the adapter for the tool signature it was given during register. When we registered the application, we persisted the tool signature (along with the SPF URL and Plant against our internal plant.) Since we have passed the plant over in the tool parameters, it is a single line of code to return the tool signature. Just like this…

```
Return m_plant.SpfSignature
```

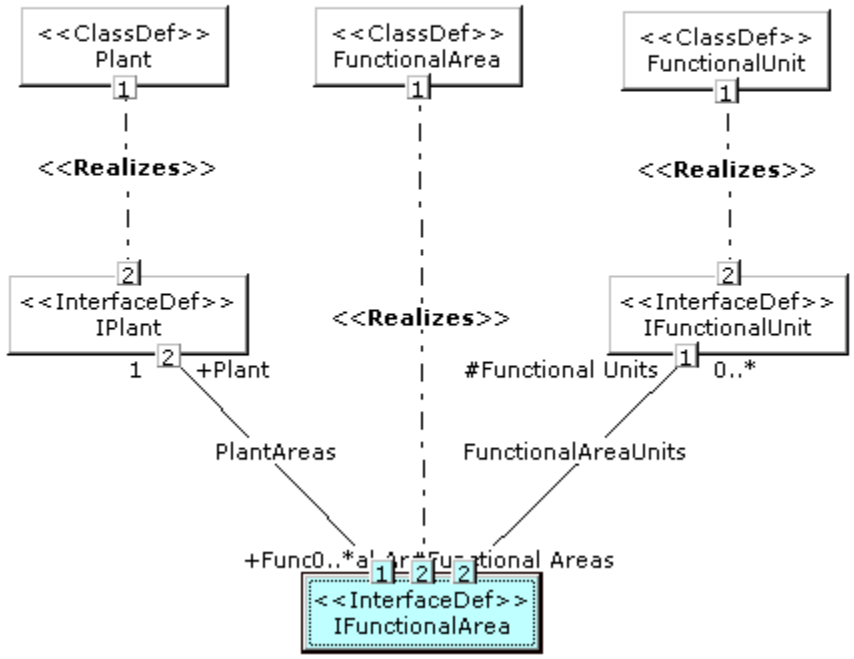
11.3.5. Implement the RetrieveDocument Method

For each document selected by the user in the retrieve dialog, the SmartPlant Client will call the RetrieveDocument method on the adapter passing it all the containers (data, metadata, and instructions). At this point in the course, we're only going to be concerned with retrieving from the data container. (More on the other containers later.)

What we want to do in this method is simply loop through all the objects in the data container and retrieve what we can understand. Let's take a look at the schema for the plant breakdown structure we'll retrieve.

Note: The following graphic shows the schema for a PBS of Plant, Area, and Unit. In a production system, this structure may be redefined so an application should be prepared to adapter to a different schema or it will place constraints on the end users.

Class Diagram for PBS



This diagram shows three ClassDefs (Plant, FunctionalArea, and FunctionalUnit) that forms the entities of the PBS with two relationships between them (PlantAreas and FunctionalAreaUnits). There are two things to make note of for this schema…

1. The RelDefs shown here are actually specializations of the abstract RelDef PBSItemCollection. This means that your adapter can treat all the PBS rels the same; using PBSItemCollection.
2. The PBSItemCollection RelDef is defined with the parent object as End2 and the child objects as End1. This will be very important when using the APIs to navigate that relationship.

To loop through all the data objects in the container is easy…

```
For Each oIObject As IObject In dataContainer.ContainedObjects
```

Next, we need to act upon those objects we can understand. We'll put in a Case statement to act based on the ClassDef of the incoming data object…

```
Select Case oIObject.ClassDefIObj.UID
    Case "Plant"
        pbsEntity = m_plant
    Case "FunctionalArea"
        pbsEntity = New Sys7Area()
    Case "FunctionalUnit"
        pbsEntity = New Sys7Unit()
    Case Else
        pbsEntity = Nothing
End Select
```

This case statement takes care of the create case for Areas and Units; that is, a new object. If you notice however, the plant is always reused. That is because our internal plant is connected to the SPF plant explicitly during the Register process.

Remember the correlation algorithm presented earlier in this chapter? We now need to implement that algorithm as part of our retrieve process. Here is some code to do that…

```
            If Not pbsEntity.ReadBySameAs(m_db, oIObject.UID) Then
                If Not pbsEntity.ReadByName(m_db, oIObject.Name)
Then
                    ' Nothing to do, this just means we cannot
correlate so it is a create
                End If

                ' Either way, we're now correlated with the
incoming object so set the SameAsId
                pbsEntity.SameAsId = oIObject.UID
            End If
```

The first statement is a call to pbsEntity.ReadBySameAs. We persist correlation by writing the incoming UID into the SameAsId property on our internal object. This call attempts to read the existing values out of the database if it can find a matching UID in that column (property).

If the call to ReadBySameAs is unsuccessful, then we try to correlate by name using the ReadByName function. If that succeeds, the existing values are read from the database.

If we have just correlated by name, this is a new correlation so we save away the SameAsId.

Now, we need to retrieve the property values for our object. Here's some code to retrieve the Name and Description properties…

```
            If oIObject.ClassDefIObj.UID <> "Plant" Then
pbsEntity.Name = oIObject.Name
            If Not oIObject.Description Is Nothing Then
pbsEntity.Description = oIObject.Description
```

We treat Plant special because most apps do not force the internal name of the plant to be exactly like the name of the SPF plant.

To retrieve additional properties would just be to add in additional lines of code although properties not on IObject would need to be accessed using GetInterface and GetProperty.

To store these objects to our design database, we just need to call the Write function on the object…

```
pbsEntity.Write(m_db)
```

However, before we do that, we'd like to retrieve the relationships between the Plant and Areas and the Area and Units. Our application data model persists these relationships by simply putting the parent ID on each child objects. So, you will find a property on Area called ParentId containing the Plant Id and the same mechanism between Unit and it's parent Area.

Remember, we can treat both of these relationships generically, so the code to navigate the rels is as follows…

```
                Dim parentRel As IRel =
relHelper.GetRelForDefUID(oIObject.End1IRelCollection,
"PBSItemCollection")
                If Not parentRel Is Nothing Then
                    pbsEntity.ParentId = parentRel.UID2
```

One problem, though. This gives us the incoming UID of the parent while we need to store this relationship in terms of our internal Id. We need a bit more code to find the parent by matching that UID against our SameAsId, then store the Id of that object in the ParentId property. That code is implemented by each internal class in the ResolveParentId method. You can look at that implementation in the Sys7DAL project.

That's it for retrieve. Here is the complete listing for RetrieveDocument…

```
        ' Set up some stuff we'll need later on
        Dim pbsEntity As Sys7Entity = Nothing
        Dim relHelper As IRelHelper = New Helper

        For Each oIObject As IObject In dataContainer.ContainedObjects
            Select Case oIObject.ClassDefIObj.UID
                Case "Plant"
                    pbsEntity = m_plant
                Case "FunctionalArea"
                    pbsEntity = New Sys7Area()
                Case "FunctionalUnit"
                    pbsEntity = New Sys7Unit()
                Case Else
                    pbsEntity = Nothing
            End Select

            If Not pbsEntity Is Nothing Then
                ' This means our adapter can understand this object,
so retrieve it
                ' First, see if we've already retrieved it (find it by
SameAsId)
                If Not pbsEntity.ReadBySameAs(m_db, oIObject.UID) Then
```

```
                    If Not pbsEntity.ReadByName(m_db, oIObject.Name)
Then
                        ' Nothing to do, this just means we cannot
correlate so it is a create
                    End If

                    ' Either way, we're now correlated with the
incoming object so set the SameAsId
                    pbsEntity.SameAsId = oIObject.UID
                End If

                ' Now, update the name and description and write it
back
                ' Note, tools do not force their internal plant name
to be the same
                ' as SPF's plant. We're excluding that here.
                If oIObject.ClassDefIObj.UID <> "Plant" Then
pbsEntity.Name = oIObject.Name
                pbsEntity.Description = oIObject.Description

                ' Find the parent
                Dim parentRel As IRel =
relHelper.GetRelForDefUID(oIObject.End1IRelCollection,
"PBSItemCollection")
                If Not parentRel Is Nothing Then
                    pbsEntity.ParentId = parentRel.UID2
                    pbsEntity.ResolveParentId(m_db)
                End If

                pbsEntity.Write(m_db)
            End If
        Next
    End Sub
```

11.4. Lab Exercise

Write the code to retrieve the Plant Breakdown structure into the application database.

You may use the Chapter 11 – Lab 1 folder as a starting point.

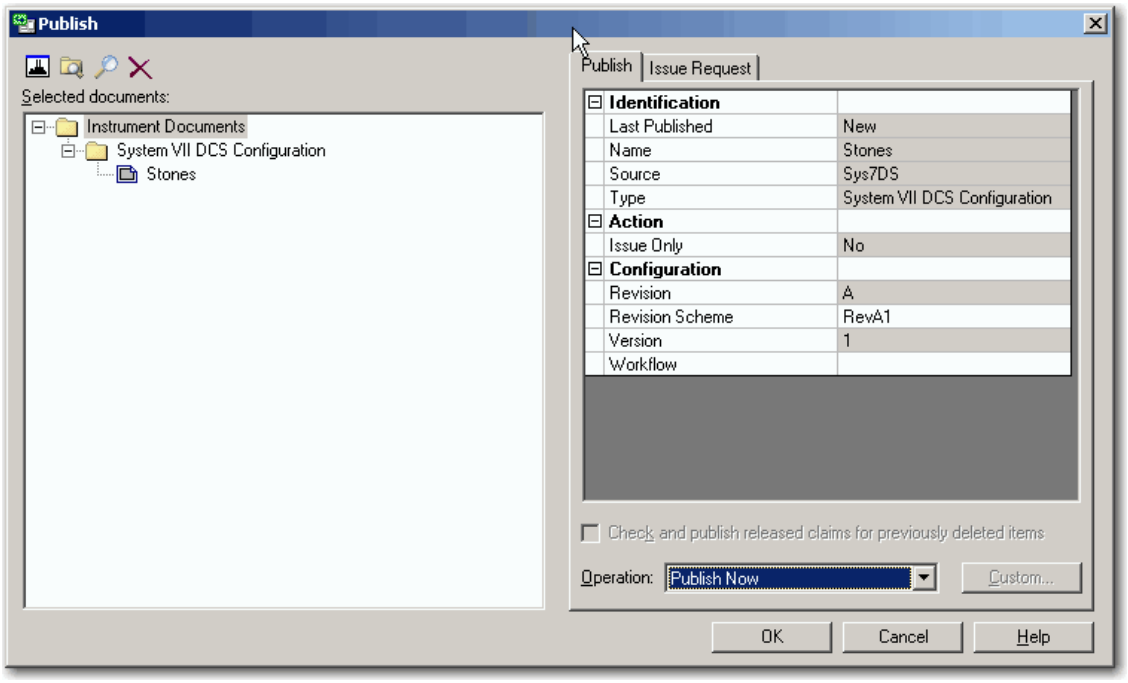
The completed lab exercise is available in the *Solution* project folder.

12. Publishing Documents

In order to publish documents to *SmartPlant*, both your application and adapter must implement code. As in the previous chapter on retrieve, your application must implement a handler for the Publish command. However, the bulk of the work is done in the adapter.

In this chapter, were going to focus on publishing a “dumb” document; i.e. one containing no data. Our example application will generate a report of IO tags for a given controller in an Excel Workbook so we will publish that as a view files. In the next chapter, we’ll build upon this and publish data.

Like retrieve, the SmartPlant Client (actually, the EF Common UI) supplies the GUI for publish…



This GUI actually does quite a few things that are beyond the scope of this document. If the application adds a document (such as a document that is active in its GUI when the user chose the Publish command), that document will be displayed in the treeview. To complete the publish of that document, you need to set the revision scheme (for the first revision only). Just click in the blank field to the right of Revision Scheme and select any schema. Click the OK button to begin the Publish process.

12.1. Code To Support Publish

The changes we need to make to support publishing a document are as follows…

- Implement the menu handler in the application for the Publish command on the SmartPlant menu.
- Tell SmartPlant that support publishing the 'System VII DCS Configuration' document type in the GetPublishableDocTypes, GetDocClassDefUID, and GetCompSchemaUID methods on the adapter.
- Implement the PublishDocument method on the adapter.

12.1.1. Implementing the Publish Menu Handler

The application code for the publish command should follow these basic steps…

- Create an instance of the SmartPlant Client.
- Create and initialize the tool parameters.
- Create an instance of our tool docs class.
- Connect to SPF.
- Get a document list container and add in active documents.
- Initiate the Publish process.

The first four steps of this process are identical to the Retrieve handler we implemented in the last chapter.

Many applications will have some scope of publish open or active when the user chooses the Publish command. For example, when the user is viewing or editing a drawing within SPP&ID and chooses Publish, it is assumed that the user wants to Publish that open drawing. Not every application will have such a concept, but if it does, that document should be placed in the document list immediately before starting the Publish process.

The first step is to get SmartPlant to give us an empty document list container scoped by the proper component schema (Generic Document Component)…

```
            Dim documentContainer As IContainer = Nothing
            result =
commonUIApplication.GetDocumentListContainerForPublish(documentContain
er)
            If result < 0 Then Throw New Exception("Error getting
document list container. [" & commonUIApplication.LastErrorMessage &
"]")
```

Then, create the an instance of the Document ClassDef within the document list container for the active document. Since the scope of our publish is a controller, the current controller (defined when a controller or one of it's children is selected in the Treeview) is 1:1 the active document. In the menu handler, we'll call a helper function CreateDocument…

```
      CreateDocument(documentContainer, controller.Id & "-Document",
controller.Name, "", "Control system configuration for " &
controller.Name, "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}")
```

Notice that the document UID is formed by adding the suffix "-Document" to the controller UID. As long as we make sure the controller UID meets all the requirements, the document UID will as well.

The last step in the Publish menu handler is to kick off the Publish process. This is done with the following…

```
            result =
commonUIApplication.ShowPublishDialog(documentContainer)
            If result < 0 Then Throw New Exception("Error during
Publish. [" & commonUIApplication.LastErrorMessage & "]")
```

The CreateDocument helper function follows the examples of creating data from Section 1 of this course; Schema Component Programming. The following is an explanation of that function…

Find the ClassDef "Document"…

```
        Dim documentClassDefIClassDef As IClassDef
        documentClassDefIClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("Document")
        If documentClassDefIClassDef Is Nothing Then Throw New
Exception("Error retrieving document classdef.")
```

Create an instance from the ClassDef…

```
        Dim documentIObject As IObject
        documentIObject =
documentClassDefIClassDef.CreateInstance(dataContainer)
        If documentIObject Is Nothing Then Throw New Exception("Error
creating document.")
```

Set properties on IObject…

```
        documentIObject.UID = uid
        documentIObject.Name = name
        If description.Length > 0 Then documentIObject.Description =
description
```

Get a reference to the IDocument interface…

```
        Dim documentIDocument As IInterface
        documentIDocument =
documentIObject.IClassDefComponent.GetInterface("IDocument", True)
        If documentIDocument Is Nothing Then Throw New
Exception("Error creating IDocument interface.")
```

Set the required properties on IDocument…

```
        Dim docCategoryIEnumHierarchy As IEnumHierarchy
        docCategoryIEnumHierarchy =
documentIDocument.GetProperty("DocCategory", True)
docCategoryIEnumHierarchy.SetHierarchyToEnumUID(documentIDocument,
typeHierarchyUid)
        If title.Length > 0 Then
documentIDocument.GetProperty("DocTitle", True).Value = title
```

The complete listing for the Publish menu handler is as follows….

```
        Try
            ' Need the registration information out of the selected
plant
            Dim plant As Sys7Plant = GetSelectedPlant()
            If plant Is Nothing Then Throw New Exception("Error
getting selected plant.")

            ' Also need to know which controller we're publishing
            Dim controller As Sys7Controller = GetSelectedController()
            If controller Is Nothing Then Throw New Exception("Error
getting selected controller.")

            ' Create an instance of the SmartPlant Common UI
            Dim commonUIApplication As IEFCommonUIApplication
            commonUIApplication =
CreateObject("EFCommonUI.Application")
            If commonUIApplication Is Nothing Then Throw New
Exception("Error starting SPF Common UI.")

            ' Set up the tool paramters
            Dim toolParameters As IEFToolParameters
            toolParameters = New EFToolParameters()
            If toolParameters Is Nothing Then Throw New
Exception("Error instancing tool parameters.")

            ' Send needed information over to the adapter through the
tool parameters
            toolParameters.Add("Database", _db, "Database")
            toolParameters.Add("Plant", plant, "Plant")

            ' Create an instance of the tool docs (even though we
won't be using it right now
            Dim toolDocs As IEFToolDocs
            toolDocs = CreateObject("Sys7Adapter.ToolDocs")
            If toolDocs Is Nothing Then Throw New Exception("Error
instancing tool docs.")

            ' Connect to SPF
            Dim result As Integer =
commonUIApplication.Connect(plant.SpfUrl, plant.SpfPlant, "", "",
"Sys7DS", "Sys7Adapter.Adapter", toolParameters, toolDocs)
            If result < 0 Then Throw New Exception("Error connecting
to SPF. [" & commonUIApplication.LastErrorMessage & "]")

            ' Ask the SmartPlant client to set up the document list
container. We'll use this to create a document for the
            ' currently selected controller.
```

```
            Dim documentContainer As IContainer = Nothing
            result =
commonUIApplication.GetDocumentListContainerForPublish(documentContain
er)
            If result < 0 Then Throw New Exception("Error getting
document list container. [" & commonUIApplication.LastErrorMessage &
"]")

            ' Go ahead and add the current controller into the list
            CreateDocument(documentContainer, controller.Id & "-
Document", controller.Name, "", "Control system configuration for " &
controller.Name, "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}")

            ' Start the publish process. The SmartPlant client takes
over from here and will make use of the adapter
            ' during the process.

            result =
commonUIApplication.ShowPublishDialog(documentContainer)
            If result < 0 Then Throw New Exception("Error during
Publish. [" & commonUIApplication.LastErrorMessage & "]")

        Catch exp As Exception
            MessageBox.Show("An error occurred during the publish
process. " & exp.Message, "Publish", MessageBoxButtons.OK)
        End Try
```

The complete listing for the CreateDocument helper function is as follows….

```
Public Function CreateDocument(ByVal dataContainer As IContainer,
ByVal uid As String, ByVal name As String, ByVal description As
String, ByVal title As String, ByVal typeHierarchyUid As String) As
IObject
        ' Note: documents are created early in the publish process and
only the generic document component schema
        ' has been loaded at this point. The documents are created
generically using the Document ClassDef. The
        ' SmartPlant Client will covert those later to the appropriate
ClassDef based on response from the adapter
        ' method GetDocClassDefUID
        Dim documentClassDefIClassDef As IClassDef
        documentClassDefIClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("Document")
        If documentClassDefIClassDef Is Nothing Then Throw New
Exception("Error retrieving document classdef.")

        ' From the classdef, we can create an instance
        Dim documentIObject As IObject
        documentIObject =
documentClassDefIClassDef.CreateInstance(dataContainer)
        If documentIObject Is Nothing Then Throw New Exception("Error
creating document.")

        ' UID, name, and description are hard-code properties on
IObject
```

```
        documentIObject.UID = uid
        documentIObject.Name = name
        If description.Length > 0 Then documentIObject.Description =
description

        ' Other required properties are on the IDocument interface,
whcih is not hard-coded
        Dim documentIDocument As IInterface
        documentIDocument =
documentIObject.IClassDefComponent.GetInterface("IDocument", True)
        If documentIDocument Is Nothing Then Throw New
Exception("Error creating IDocument interface.")

        ' The easiest way to deal with an enum hierarchy is using the
IEnumHierarchy. This call will
        ' set all level properties based on the bottom value. We don't
need to worry about whether a 2nd
        ' or 3rd level value was passed in.
        Dim docCategoryIEnumHierarchy As IEnumHierarchy
        docCategoryIEnumHierarchy =
documentIDocument.GetProperty("DocCategory", True)

docCategoryIEnumHierarchy.SetHierarchyToEnumUID(documentIDocument,
typeHierarchyUid)

        ' Title is another required property on IDocument
        If title.Length > 0 Then
documentIDocument.GetProperty("DocTitle", True).Value = title

        ' Return the new document in case the caller wants to do more.
But at this point, it is in
        ' the data container ready to go.
        CreateDocument = documentIObject
    End Function
```

12.1.2. Implementing GetPublishableDocTypes and GetDocClassDefUID, & GetCompSchemaUID

Once the publish process has been started, the SmartPlant Client is going to ask the adapter for some information regarding it's publish capabilities. It needs to know what kinds of documents the adapter can publish. It will call the GetPubishableDocTypes method on the adapter and expect it to return a collection of document types in an instance of the class EFDocTypes.

To implement this method in our adapter, we simply need to create an instance of EFDocTypes and then add in each of the document type UIDs that we can publish. This code is very similar to the code we added to GetRetrieveableDocTypes except that we will only publish the document type 'System VII DCS Configuration'.

```
        Dim docTypes As IEFDocTypes = New EFDocTypes
        If docTypes Is Nothing Then Throw New Exception("Error
creating publishable document types.")
```

```
        ' Add the System VII DCS Configuration
        docTypes.Add("{64CBA09B-50D3-4FF2-8EE3-7725871F8728}")
        GetPublishableDocTypes = docTypes
```

In the last chapter, we implemented GetDocClassDefUID to return the correct ClassDef for each document type. We need to add in the case of our publishable document type 'System VII DCS Configuration'…

```
        GetDocClassDefUID = Nothing
        Select Case sDocTypeUID
            Case "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}"
                ' System VII DCS Configuration
                GetDocClassDefUID = "CSDocument"
            Case "{E4528C43-83A9-11D6-BD6F-00104BCC2B69}"
                ' PBS document
                GetDocClassDefUID = "PBSDocument"
            Case "{F84347CF-A57A-4CBE-894A-2F5B2ED1EE56}"
                ' SPI DCS Configuration
                GetDocClassDefUID = "CSDocument"
        End Select
```

The SmartPlant client will also want to know the component schema to be used for each document type. When it prepares the containers for each of these documents, it will only load that component schema to scope the data…

```
        GetCompSchemaUID = Nothing
        Select Case sDocTypeUID
            Case "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}"
                ' System VII DCS Configuration
                GetCompSchemaUID = "ControlSystemComponent"
            Case "{E4528C43-83A9-11D6-BD6F-00104BCC2B69}"
                ' PBS document
                GetCompSchemaUID = "PBSComponent"
            Case "{F84347CF-A57A-4CBE-894A-2F5B2ED1EE56}"
                ' SPI DCS Configuration
                GetCompSchemaUID = "ControlSystemComponent"
        End Select
```

12.1.3. Implement the PublishDocument Method

Since we're not going to be publishing data with this document, the requirements for the PublishDocument methods are very simple. In fact, all we need to do is create an instance of the ClassDef File in the metadata container.

We did not pass the active controller from the application through the ToolParameters, but the document is a function of the controller. Remember the document UID is the controller UID with "-Document" appended. We can reverse that process to determine the controller we're publishing…

```
        Dim controllerId As String =
ControllerIdFromDocumentId(oDocumentIObj.UID)
        Dim controller As Sys7Controller = New Sys7Controller()
```

```
        If Not controller.ReadById(m_db, controllerId) Then Throw New
Exception("Error finding controller.")
```

Next, let's regenerate the Excel controller report. We have a helper function called GenerateViewFile implemented to do this for us…

```
        Dim filePath As String = ""
        Dim fileName As String = ""
        GenerateViewFile(filePath, fileName, controller)
```

Finally, we need to create the file object in the meta data container. We'll use a helper function…

```
        CreateFile(oDocMetaIContainer, filePath, fileName)
```

Here is the complete listing of the implementation of PublishDocument…

```
        ' Go get the controller associated with this document
        Dim controllerId As String =
ControllerIdFromDocumentId(oDocumentIObj.UID)
        Dim controller As Sys7Controller = New Sys7Controller()
        If Not controller.ReadById(m_db, controllerId) Then Throw New
Exception("Error finding controller.")

        ' Need to generate the view file for this publish
        Dim filePath As String = ""
        Dim fileName As String = ""
        GenerateViewFile(filePath, fileName, controller)

        ' Create the file object in the metaContainer
        CreateFile(oDocMetaIContainer, filePath, fileName)
```

The following is a complete listing of the CreateFile helper function. There should be nothing new in this method from Section 1 of this course…

```
        ' Start by finding the File ClassDef
        Dim fileClassDefIClassDef As IClassDef =
metaContainer.ContainerComposition.GetIObjectForUID("File")
        If fileClassDefIClassDef Is Nothing Then Throw New
Exception("Error getting file classdef.")

        ' Create the file object
        Dim fileIObject As IObject =
fileClassDefIClassDef.CreateInstance(metaContainer)
        If fileIObject Is Nothing Then Throw New Exception("Error
creating file instance.")

        ' Set the IObject properties. Note that file UID should change
with each version so we can generate a
        ' new one with each publish.
        fileIObject.UID =
metaContainer.ContainerComposition.GenerateUID()
        fileIObject.Name = fileName
```

```
        ' There are more required properties on the IFile interface
        Dim fileIFile As IInterface =
fileIObject.IClassDefComponent.GetInterface("IFile", True)
        If fileIFile Is Nothing Then Throw New Exception("Error
creating IFile interface.")
        fileIFile.GetProperty("FilePath", True).Value = filePath
        fileIFile.GetProperty("FileType", True).Value = "ExcelFile"

        ' The SmartPlant Client has created the document version in
the meta container for us. We need to find
        ' it and create a rel between it and our file.
        Dim versionsIObjCollection As IObjectCollection =
metaContainer.ContainerComposition.GetInstancesForClassDef("DocumentVe
rsion")
        If versionsIObjCollection Is Nothing Then Throw New
Exception("Error finding document version.")
        If versionsIObjCollection.Count <> 1 Then Throw New
Exception("Error finding document version.")

        ' Should be one and only one doc version in this collection
        Dim versionIObject As IObject = versionsIObjCollection.Item(1)
        If versionIObject Is Nothing Then Throw New Exception("Error
finding document version.")

        ' Need a rel helper
        Dim relHelper As IRelHelper = New Helper()
        If relHelper Is Nothing Then Throw New Exception("Error
creating rel helper.")

        ' Create the rel
        relHelper.CreateRelationship(metaContainer,
metaContainer.ContainerComposition.GenerateUID(), fileIObject,
versionIObject, "FileComposition")

        ' Return the file object in case caller needs to do more work
with it
        CreateFile = fileIObject
```

12.2. Lab Exercise

Implement the code requirements covered in this section in the application tool and in the adapter. Write the application code first so that you are to the point where the common *Publish* dialog is displaying and the *Engineering Tool* button displays the *Select Documents* GUI properly.

Note: Connecting to SPF is required when *ShowPublishDialog* is called; key in the username *updateuser* to connect.

You may use the Chapter 12 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

Note: When you are ready to actually publish the document, you are required to select a *Revision Scheme*.

13. Publishing Data

Now that you have learned how to publish a document (containing a *File* object) to *SmartPlant*, the next step is to publish data. The process is very similar—the main difference is that you add data objects to the data container. In the adapter, you may add code to the *PublishDocument* routine to add the data objects to be published.

The code is appended to our existing *PublishDocument* method, right after adding the relationship between the version and file objects:

```
' Publish the data
PublishController(oDocContIContainer, controller)
```

This helper function takes as arguments the data container and the controller object. Using standard Schema Component programming, we'll add an instance of the ClassDef CSController to the data container. This code is going to look very similar to the code for creating a document. First, get a reference to the ClassDef…

```
        Dim controllerClassDef As IClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("CSController")
        If controllerClassDef Is Nothing Then Throw New
Exception("Error finding cotnroller class def.")
```

Then create an instance from the ClassDef…

```
        Dim controllerIObject As IObject =
controllerClassDef.CreateInstance(dataContainer)
        If controllerIObject Is Nothing Then Throw New
Exception("Error creating controller.")
```

Set the properties on IObject…

```
        controllerIObject.UID = controller.Id
        controllerIObject.Name = controller.Name
        If controller.Description.Length > 0 Then
controllerIObject.Description = controller.Description
```

Get the IEquipment interface…

```
        Dim controllerIEquipment As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IEquipment", True)
        If controllerIEquipment Is Nothing Then Throw New
Exception("Error creating IEquipment interface.")
```

Set the equipment type hierarchy on the IEquipment interface…

```
        Dim eqTypeIEnumHierarchy As IEnumHierarchy =
controllerIEquipment.GetProperty("EqType0", True)
        If eqTypeIEnumHierarchy Is Nothing Then Throw New
Exception("Error setting eq type hierarchy.")

eqTypeIEnumHierarchy.SetHierarchyToEnumUID(controllerIEquipment,
"{7AF4EBC3-6B53-43C6-8B64-A29E33EA54F5}")
```

Instance a couple of optional interfaces (IProcessController and IPBSItem) that we'll want later on…

```
        Dim controllerIProcessController As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IProcessController"
, True)
        If controllerIProcessController Is Nothing Then Throw New
Exception("Error creating IProcessController interface.")

        Dim controllerIPBSItem As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IPBSItem", True)
        If controllerIPBSItem Is Nothing Then Throw New
Exception("Error creating IPBSItem interface.")
```

Now, we want to publish all the IO cards that are children of the controller. The Sys7DAL has a function on each parent class called ReadChildren to this for us so all we need to do is call this from the controller. It will return a collection of child IO cards…

```
        Dim ioCards As ICollection = controller.ReadChildren(m_db)
        If ioCards Is Nothing Then Throw New Exception("Error getting
io cards associated with the controller.")
```

Lastly, we'll look through the IO card and call a helper function to publish each…

```
        For Each ioCard As Sys7IOCard In ioCards
            Dim ioCardIObject As IObject =
PublishIOCard(dataContainer, ioCard)
        Next
```

The complete listing for PublishController is as follows…

```
    Private Function PublishController(ByVal dataContainer As
IContainer, ByVal controller As Sys7Controller) As IObject
        ' Start by finding the controller class def.
        Dim controllerClassDef As IClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("CSController")
        If controllerClassDef Is Nothing Then Throw New
Exception("Error finding cotnroller class def.")

        ' From the class def, create an instance of the controller
```

```
        Dim controllerIObject As IObject =
controllerClassDef.CreateInstance(dataContainer)
        If controllerIObject Is Nothing Then Throw New
Exception("Error creating controller.")

        ' Set the IObject properties
        controllerIObject.UID = controller.Id
        controllerIObject.Name = controller.Name
        If controller.Description.Length > 0 Then
controllerIObject.Description = controller.Description

        ' IEquipment
        Dim controllerIEquipment As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IEquipment", True)
        If controllerIEquipment Is Nothing Then Throw New
Exception("Error creating IEquipment interface.")

        ' Equipment type hierarchy
        Dim eqTypeIEnumHierarchy As IEnumHierarchy =
controllerIEquipment.GetProperty("EqType0", True)
        If eqTypeIEnumHierarchy Is Nothing Then Throw New
Exception("Error setting eq type hierarchy.")

eqTypeIEnumHierarchy.SetHierarchyToEnumUID(controllerIEquipment,
"{7AF4EBC3-6B53-43C6-8B64-A29E33EA54F5}")

        ' IProcessController is not a required realizes and we want
that
        Dim controllerIProcessController As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IProcessController"
, True)
        If controllerIProcessController Is Nothing Then Throw New
Exception("Error creating IProcessController interface.")

        ' Also need IPBSItem
        Dim controllerIPBSItem As IInterface =
controllerIObject.IClassDefComponent.GetInterface("IPBSItem", True)
        If controllerIPBSItem Is Nothing Then Throw New
Exception("Error creating IPBSItem interface.")

        ' Get all the IO cards associated with this controller
        Dim ioCards As ICollection = controller.ReadChildren(m_db)
        If ioCards Is Nothing Then Throw New Exception("Error getting
io cards associated with the controller.")

        ' Loop through and publish each
        For Each ioCard As Sys7IOCard In ioCards
            Dim ioCardIObject As IObject =
PublishIOCard(dataContainer, ioCard)
        Next

        PublishController = controllerIObject
    End Function
```

The complete listing for PublishIOCard is as follows…

```
    Private Function PublishIOCard(ByVal dataContainer As IContainer,
ByVal ioCard As Sys7IOCard) As IObject
        ' Start by finding the iocard class def.
        Dim ioCardClassDef As IClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("CSIOCard")
        If ioCardClassDef Is Nothing Then Throw New Exception("Error
finding classdef for IO card.")

        ' From the class def, create an instance of the io card
        Dim ioCardIObject As IObject =
ioCardClassDef.CreateInstance(dataContainer)
        If ioCardIObject Is Nothing Then Throw New Exception("Error
creating io card.")

        ' Set the IObject properties
        ioCardIObject.UID = ioCard.Id
        ioCardIObject.Name = ioCard.Name
        If ioCard.Description.Length > 0 Then
ioCardIObject.Description = ioCard.Description

        ' IEquipment
        Dim ioCardIEquipment As IInterface =
ioCardIObject.IClassDefComponent.GetInterface("IEquipment", True)
        If ioCardIEquipment Is Nothing Then Throw New Exception("Error
creating IEquipment interface.")

        ' Equipment type hierarchy
        Dim eqTypeIEnumHierarchy As IEnumHierarchy =
ioCardIEquipment.GetProperty("EqType0", True)
        If eqTypeIEnumHierarchy Is Nothing Then Throw New
Exception("Error setting eq type hierarchy.")
        eqTypeIEnumHierarchy.SetHierarchyToEnumUID(ioCardIEquipment,
"{253B45CF-3F23-4729-9B25-145C70F73BB3}")

        ' We want the IIOCard interface
        Dim ioCardIIOCard As IInterface =
ioCardIObject.IClassDefComponent.GetInterface("IIOCard", True)
        If ioCardIIOCard Is Nothing Then Throw New Exception("Error
creating IIOCard interface.")

        ' Get all the channels associated with this controller
        Dim channels As ICollection = ioCard.ReadChildren(m_db)
        If channels Is Nothing Then Throw New Exception("Error getting
IO card channels.")

        ' Loop through each and publish
        For Each channel As Sys7Channel In channels
            Dim channelIObject As IObject =
PublishChannel(dataContainer, channel)
        Next

        PublishIOCard = ioCardIObject
    End Function
```

The PublishIOCard function calls a helper function (PublishChannel) to publish the child channels…

```
    Private Function PublishChannel(ByVal dataContainer As IContainer,
ByVal channel As Sys7Channel) As IObject
        ' Need the Channel class def
        Dim channelClassDef As IClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("CSSignalChannel")
        If channelClassDef Is Nothing Then Throw New Exception("Error
finding channel class def.")

        ' Creating an instance of the channel
        Dim channelIObject As IObject =
channelClassDef.CreateInstance(dataContainer)
        If channelIObject Is Nothing Then Throw New Exception("Error
creating channel")

        ' Set the IObject properties
        channelIObject.UID = channel.Id
        channelIObject.Name = channel.Name
        If channel.Description.Length > 0 Then
channelIObject.Description = channelIObject.Description

        ' Need ISignalPort interface
        Dim channelISignalPort As IInterface =
channelIObject.IClassDefComponent.GetInterface("ISignalPort", True)
        If channelISignalPort Is Nothing Then Throw New
Exception("Error creating ISignalPort interface.")

        ' Get the IO Tags associated with this channel
        Dim ioTags As ICollection = channel.ReadChildren(m_db)
        If ioTags Is Nothing Then Throw New Exception("Error getting
io tags for channel.")

        ' Loop through each and publish
        For Each ioTag As Sys7IOTag In ioTags
            Dim ioTagIObject As IObject = PublishIOTag(dataContainer,
ioTag)
            If ioTagIObject Is Nothing Then Throw New Exception("Error
publishing io tag.")
        Next

        PublishChannel = channelIObject
    End Function
```

And finally, the PublishIOTag function publishes the IO tags…

```
    Private Function PublishIOTag(ByVal dataContainer As IContainer,
ByVal ioTag As Sys7IOTag) As IObject
        ' Need the iotag class def
        Dim ioTagClassDef As IClassDef =
dataContainer.ContainerComposition.GetIObjectForUID("CSControlSysIO")
        If ioTagClassDef Is Nothing Then Throw New Exception("Error
finding io tag class def.")
```

```
        ' Create an instance
        Dim ioTagIObject As IObject =
ioTagClassDef.CreateInstance(dataContainer)
        If ioTagIObject Is Nothing Then Throw New Exception("Error
creating ioTag.")

        ' Set the IObject properties
        ioTagIObject.UID = ioTag.Id
        ioTagIObject.Name = ioTag.Name
        If ioTag.Description.Length > 0 Then ioTagIObject.Description
= ioTag.Description

        ' Want the ILimits_AlarmOrTrip interface
        Dim ioTagILimits As IInterface =
ioTagIObject.IClassDefComponent.GetInterface("ILimits_AlarmOrTrip",
True)
        If ioTagILimits Is Nothing Then Throw New Exception("Error
creating ILimits_AlarmOrTrip interface.")

        ' Want the IScaledInstrument interface
        Dim ioTagIScaledInstrument As IInterface =
ioTagIObject.IClassDefComponent.GetInterface("IScaledInstrument",
True)
        If ioTagIScaledInstrument Is Nothing Then Throw New
Exception("Error creating IScaledInstrument interface.")

        ' Want the IDCSScaledInstrument interface
        Dim ioTagIDCSScaledInstrument As IInterface =
ioTagIObject.IClassDefComponent.GetInterface("IDCSScaledInstrument",
True)
        If ioTagIDCSScaledInstrument Is Nothing Then Throw New
Exception("Error creating IDCSScaledInstrument interface.")

        PublishIOTag = ioTagIObject
    End Function
```

13.1. Lab Exercise

Implement the code requirements covered in this section in the adapter.

Note: *There is no need to add code to the application. From the application perspective, publishing a document only looks the same as publishing plenty of data. All changes resulting from this lab are within the PublishDocument method on the adapter and the addition of some helper functions.*

You may use the Chapter 13 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

14. Publishing Additional Relationships

In the last chapter, we published all the data related to a controller: the controller, the child IO cards, the channels on the IO cards, and the IO tags connected to those channels. However, we did not publish the relationships between those objects.

Another requirement is to publish a relationship to link that entire structure in the plant breakdown structure.

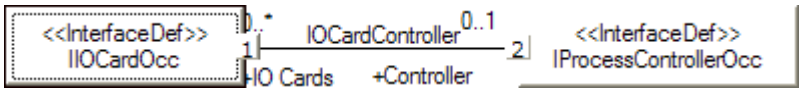
And finally, it is customary to publish a relationship between the published document and the main objects (or at least the top level object) to support navigating from a published document to the data contained within that publish.

In this chapter, we will implement all of the above. The complete list of tasks for this chapter is as follows…

- Publish the relationships between the control system objects…
    - A relationship between each IO card and it's controller.
    - A relationship between each channel and it's IO card.
    - A relationship between each IO tag and the channel it's connected to.
- Publish a relationship between the PBS and the controller.
- Publish a relationship between the document and the controller.

14.1. Controller to IO Card Rel

The following diagram shows the IOCardController relationship that we need to publish…



The first thing to notice is that the rel is between the IIOCardOcc InterfaceDef and the IProcessControllerOcc InterfaceDef. If the Realizes for your ClassDefs are optional, you must explicitly instance these interfaces using the GetInterface method. If you publish a relationship between two objects and either of them are missing an interface from the RelDef, your publish will fail with a schema validation error.

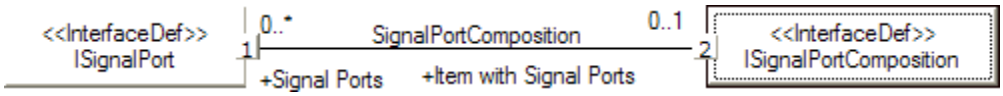
The second thing to notice is that the IO card is on end 1 and the controller is on end 2. You must take care when creating this relationships to pass the objects or UIDs in correctly.

In the PublishController helper function, we'll add one line of code to publish this relationship with each IO card…

```
        For Each ioCard As Sys7IOCard In ioCards

            Dim ioCardIObject As IObject =
PublishIOCard(dataContainer, ioCard)


            ' Create a relationship between the iocard and the
controller

            relHelper.CreateRelationship(dataContainer, ioCard.Id &
controller.Id, ioCardIObject, controllerIObject, "IOCardController")

        Next
```

14.2. IO Card to Channel Rel

The following diagram shows the IOCardController relationship that we need to publish…



Now, this RelDef is a little harder to decipher. If you think of the IO card as a collection of input and output channels, you get closer. Each channel is carrying a signal from an instrument in the field, which are collected at the IO card and converted into digital data. The IO card is a signal port composition and the channel is a signal port. (Take our word for it.)

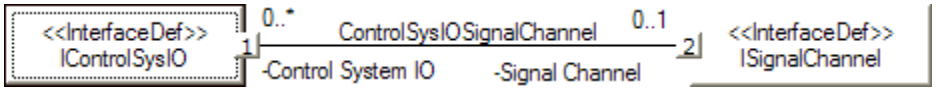
Again, you need to take notice of two things: First, make sure that your IO card has the the ISignalPortComposition interface and your channel has the ISignalPort interface. Second, notice that again the child (channel) is end 1 and the parent (IO card) is end 2 and create the relationship accordingly.

In the PublishIOCard helper function, we'll add one line of code to publish this relationship with each channel…

```
        For Each channel As Sys7Channel In channels
            Dim channelIObject As IObject =
PublishChannel(dataContainer, channel)

            ' Create a relationship between the ioCard and channel
            relHelper.CreateRelationship(dataContainer, channel.Id &
ioCard.Id, channelIObject, ioCardIObject, "SignalPortComposition")
        Next
```

14.3. Channel to IO Tag Rel

The following diagram shows the ControlSysIOSignalChannel relationship that we need to publish…



Once again, take notice of two things: First, make sure that your channel has the ISignalChannel interface and your IO tag has the IControlSysIO interface. Second, notice that again the child (IO tag) is end 1 and the parent (channel) is end 2 and create the relationship accordingly.

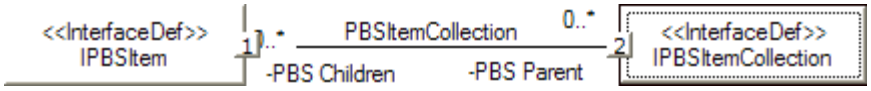
In the PublishChannel helper function, we'll add one line of code to publish this relationship with each IO tag…

```
        For Each ioTag As Sys7IOTag In ioTags
            Dim ioTagIObject As IObject = PublishIOTag(dataContainer,
ioTag)
            If ioTagIObject Is Nothing Then Throw New Exception("Error
publishing io tag.")

            ' Create a relationship between the io tag and the channel
            relHelper.CreateRelationship(dataContainer, ioTag.Id &
channel.Id, ioTagIObject, channelIObject, "ControlSysIOSignalChannel")
        Next
```

14.4. Relate the Controller into the PBS

The convention is that each application should relate, at least it's top level items, into the PBS by publishing a relationship to the bottom level in that structure. In our example, we will assume that to be the Unit.

The following diagram shows the PBSItemCollection that is defined for this purpose…



Again, make sure that your Controller has the IPBSItem and take note that the child (controller) is end 2 and the parent (unit) is end 1.

However, the application should not publish the Unit itself; only SPF published PBS. So, how do you get a relationship past schema validation when one end is not present in the data container? This is possible because this RelDef is defined such that end2 may not be present in this container. The schema validation code will not create an error when it finds that object missing and will assume that it has been published by another application.

To implement this, we'll add some code to PublishController. First, go get the associate Unit from our database. Remember the ParentId is a property on the controller…

```
        Dim unit As Sys7Unit = New Sys7Unit()
        If Not unit.ReadById(m_db, controller.ParentId) Then Throw New
Exception("Error finding pbs parent for controller.")
```

Next, create the relationship. One thing that is unusual here. We don't use our internal Id for the Unit because SmartPlant Foundation doesn't know anything about that Id; we will never publish that object. We need to use the SPF UID when publishing that rel, which is persisted in the SameAsId of the Unit…

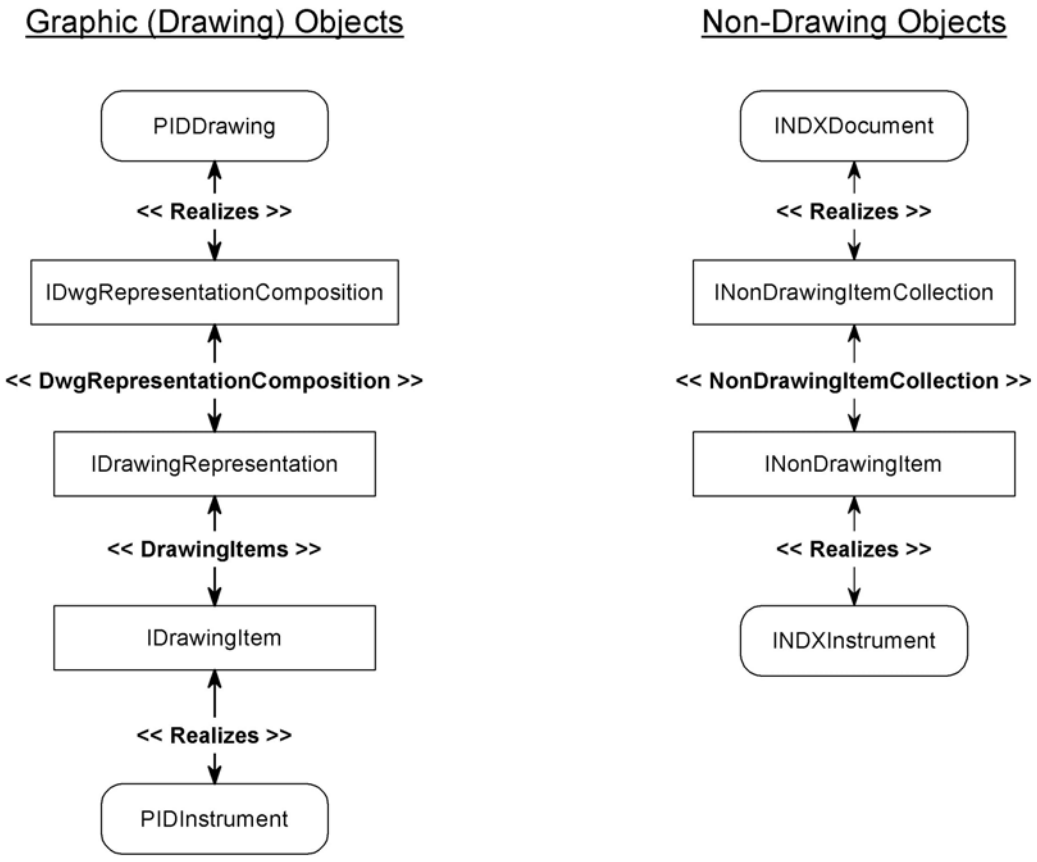
```
        relHelper.CreateRelationship(dataContainer, controller.Id & "-
PBSItemCollection", controllerIObject, unit.SameAsId,
"PBSItemCollection")
```

14.5. Relate the Controller to the Document

There are actually two ways to do this based on whether the document is graphical (drawing) or non-graphical.

In the left diagram below, you see the structure that is pubished between a P&I diagram and instruments on that drawing. The graphical object-to-document relationship is desirable to support data navigation in the SPF Client via *hotspotting* from within a graphics file. There are two RelDefs involved along with a drawing representation object in the middle. The drawing representation object has a property to carry the graphic object ID (or GOID). The SmartPlant Loader actually builds a graphical map of the drawing; associating each object UID with the GOID. SmartPlant Markup uses this to support graphical navigation.

The right diagram below show the structure published between an instrument index and the contained instruments. The non-graphical object-to-document relationship (NonDrawingItemCollection) allows the user to right-mouse click on a document in the SPF Client to view non-drawing items.



Because our sample application is non-graphical, we will publish data that is associated with the *Sys7DS* document, which realizes an interface called *INonDrawingItemCollection*, denoting its association with non-graphical data. The

data objects we publish will support *INonDrawingItem* in order to participate in the required relationship between document and object, which is called *NonDrawingItemCollection*.

We can modify our example adapter's *PublishController* routine to process the Sys7DS data as non-drawing items…

```
        relHelper.CreateRelationship(dataContainer, controller.Id & "-
NonDrawingItemCollection", controllerIObject, controller.Id & "-
Document", "NonDrawingItemCollection")
```

14.6. Lab Exercise

Implement the code requirements covered in this section in the adapter.

Note: *There is no need to add code to the application. From the application perspective, publishing a document only looks the same as publishing plenty of data. All changes resulting from this lab are within the PublishDocument method on the adapter and the addition of some helper functions.*

You may use the Chapter 14 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

15. Retrieving Control System Data

In Chapter 11, we retrieved the Plant Breakdown Structure from SmartPlant Foundation. For the most part, the code to support that retrieve will provide most of the support to retrieve the control system data. In fact, the only changes we need to make are in the RetrieveData method of the adapter. It was coded to consume Plants, Areas, and Units and we need to add functionality to consume controllers, IO cards, channels, and IO tags.

The first thing to do is expand our Select statement to encompass the additional ClassDefs we can expect…

```
Select Case oIObject.ClassDefIObj.UID
    Case "Plant"
        pbsEntity = m_plant
    Case "FunctionalArea"
        pbsEntity = New Sys7Area()
    Case "FunctionalUnit"
        pbsEntity = New Sys7Unit()
    Case "CSController"
        pbsEntity = New Sys7Controller()
    Case "CSIOCard"
        pbsEntity = New Sys7IOCard()
    Case "CSSignalChannel"
        pbsEntity = New Sys7Channel()
    Case "CSControlSysIO"
        pbsEntity = New Sys7IOTag()
    Case Else
        pbsEntity = Nothing
End Select
```

The correlation algorithm is exactly the same so there is no need to change any of that. Also, we're only retrieving the properties on IObject and those are common to all ClassDefs so there is no need to change that.

We do have a problem following the retrieving the parent relationship. If you recall back in Chapter 11, we had a single statement to navigate all the parent relationships because they were all PBSItemCollection…

```
        Dim parentRel As IRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
```

This will not work now, because the relationships between the control system objects are all different. Let's change the line of code above to call a helper function…

```
        pbsEntity.ParentId = GetParentId(oIObject)
```

This helper function will then navigate the appropriate relationship depending on the ClassDef of the child object…

```
    Select Case oIObject.ClassDefIObj.UID
            Case "Plant"
                Return Nothing
            Case "FunctionalArea"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "FunctionalUnit"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "CSController"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "CSIOCard"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "IOCardController")
            Case "CSSignalChannel"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "SignalPortComposition")
            Case "CSControlSysIO"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "ControlSysIOSignalChannel")
        End Select
```

The complete listing for the RetrieveData method is as follows…

```
    Private Sub RetrieveData(ByVal dataContainer As IContainer)
        ' Set up some stuff we'll need later on
        Dim pbsEntity As Sys7Entity = Nothing
        Dim relHelper As IRelHelper = New Helper

        For Each oIObject As IObject In dataContainer.ContainedObjects
            Select Case oIObject.ClassDefIObj.UID
                Case "Plant"
                    pbsEntity = m_plant
                Case "FunctionalArea"
                    pbsEntity = New Sys7Area()
                Case "FunctionalUnit"
                    pbsEntity = New Sys7Unit()
                Case "CSController"
                    pbsEntity = New Sys7Controller()
                Case "CSIOCard"
                    pbsEntity = New Sys7IOCard()
                Case "CSSignalChannel"
                    pbsEntity = New Sys7Channel()
                Case "CSControlSysIO"
                    pbsEntity = New Sys7IOTag()
                Case Else
                    pbsEntity = Nothing
            End Select

            If Not pbsEntity Is Nothing Then
                ' This means our adapter can understand this object,
so retrieve it
                ' First, see if we've already retrieved it (find it by
SameAsId)
                If Not pbsEntity.ReadBySameAs(m_db, oIObject.UID) Then
```

```
                    If Not pbsEntity.ReadByName(m_db, oIObject.Name)
Then
                        ' Nothing to do, this just means we cannot
correlate so it is a create
                    End If

                    ' Either way, we're now correlated with the
incoming object so set the SameAsId
                    pbsEntity.SameAsId = oIObject.UID
                End If

                ' Now, update the name and description and write it
back
                ' Note, tools do not force their internal plant name
to be the same
                ' as SPF's plant. We're excluding that here.
                If oIObject.ClassDefIObj.UID <> "Plant" Then
pbsEntity.Name = oIObject.Name
                pbsEntity.Description = oIObject.Description

                ' Find the parent
                pbsEntity.ParentId = GetParentId(oIObject)
                If (pbsEntity.ParentId.Length() > 0) Then
                    pbsEntity.ResolveParentId(m_db)
                End If

                pbsEntity.Write(m_db)
            End If
        Next
    End Sub
```

The complete listing for the GetParentId method is as follows…

```
    Private Function GetParentId(ByVal oIObject As IObject) As String
        ' Need a rel helper
        Dim relHelper As IRelHelper = New Helper()

        Dim parentRel As IRel = Nothing
        Select Case oIObject.ClassDefIObj.UID
            Case "Plant"
                Return “”
            Case "FunctionalArea"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "FunctionalUnit"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "CSController"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "PBSItemCollection")
            Case "CSIOCard"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "IOCardController")
            Case "CSSignalChannel"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "SignalPortComposition")
```

```
            Case "CSControlSysIO"
                parentRel = relHelper.GetRelForDefUID(
oIObject.End1IRelCollection, "ControlSysIOSignalChannel")
        End Select
        If parentRel Is Nothing Then
            Return ""
        Else
            ' The reason we're returning just the UID of the parent is
that some rels may be dangling.
            ' This is exactly the case with the rel between the Unit
and the Controller. UID2IObj will be nothing
            ' in that case, so we need to act upon the Id.
            Return parentRel.UID2
        End If
    End Function
```

15.1. Lab Exercise

Add functionality into your adapter to support retrieve of control system data.

You may use the Chapter 15 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

15.2. Retrieving a View File

This chapter has dealt so far with retrieving document containers and the data associated with a document; however, when we will discussed publishing to *SmartPlant*, we published an actual file. These files are known as “View Files” in SmartPlant terminology. Retrieving a view file is a fairly straight forward process.

The first step is to let the EF Client know that you support retrieving a view file. You do this in the *SupportsFeature* routine as shown.

```
Private Function SupportsFeature(ByVal Feature As
SchemaCompInterfaces.ToolFeatures, Optional sDocTypeUID As String) As Boolean
    ‘
    Select Case Feature
         ‘// Add support for RetrieveViewFile
         Case RetrieveViewFile
           IEFAdapter_SupportsFeature = True
         Case Else
           IEFAdapter_SupportsFeature = False
    End Select
    ‘
End Function
```

When *RetrieveDocument* is called you will need to add some code to search for and save the view file. To accomplish this, search the meta data container for instances of “File” objects, get the IObject for the file, read the FilePath property and save the file as shown in the following RetrieveViewFile sub routine.

```
Private Sub RetrieveViewFile(ByVal oDocMetaContainer As IContainer)
    ‘
    Dim collObjects As IObjectCollection
    Dim oFile As IObject
    Dim oInterface As IInterface
    Dim oProperty As IProperty
    Dim vSplit
    ‘
    ‘// Search the MetaData Container for instances of “File”
    oDocMetaContainer.IContainerQuery.GetInstancesForClassDef
“File”, collObjects
    ‘
    ‘// Loop through the object collection
    For Each oFile in collObjects
        ‘// Get the IFile Interface
        Set oInterface = oFile.IClassDefComponent.GetInterface(“IFile”, False)
        ‘// Get the FilePath property
        Set oProperty = oFile.GetProperty(“FilePath”, False)
        ‘// Separate the FileName from the path
        vSplit = Split(oProperty.Value, “\”)
        ‘// Save the File to C:\Temp
        FileCopy oProperty.Value, “C:\Temp\” & vSplit(Ubound(vSplit))
    Next oFile
    ‘
End Sub
```

16. Delete Instructions

One of the benefits of SmartPlant Integration is the notification and communication of deletes. This is generally not the case with import/export functionality or even many integration technologies.

Since most applications do not remember deleted objects, SmartPlant Foundation assists in this process. Each publish is compared to the previous publish of the same document (unless of course, this is the first). Missing objects / rels are recognized and the SmartPlant Client asks the adapter what the status of the object is. This call is to the GetObjectStatusInTool or GetRelStatusInTool methods on the adapter. The adapter responds that the object is either deleted or has been moved to another container (still exists). For objects which the adapter reports as no longer in existence, the SmartPlant Client generates a delete instruction.

Any tool retrieving this document will get an instructions container along with the data and metadata containers. The retrieving tool should process these delete instructions and delete the internal objects accordingly.

In this chapter, we will do the following…

- Implement the GetObjectStatusInTool method on the adapter.
- Implement the GetRelStatusInTool method on the adapter.
- Add code to process the delete instructions into RetreiveDocument.

16.1. IDeleteInstruction

Delete Instructions provide access to the object they are referring to via the *IRefObject* property exposed on the *IDeleteInstruction* interface. This property allows you to access information about the object that the instruction applies to so that you can determine whether you need to process the object in your application.

Property	Description
DeleteTransitionIObj As IObject	Optional property (DeleteTransition). Sets/returns the enumeration object for the delete transition for the deleted object.
DeleteTransitionSetFlag As Boolean	Indicates whether DeleteTransition has been set.
DeleteTransitionText As String	Optional property (DeleteTransition). Sets/returns the text for the delete transition for the deleted object.
DeleteTransitionUID As	Optional property (DeleteTransition). Sets/returns the

Property	Description
String	UID for the delete transition for the deleted object.
IInstruction As IInstruction	Returns the implied IInstruction interface.
IRefObject As IRefObject	Returns the implied IRefObject interface.
IRefRel As IRefRel	Returns the optionally implied IRefRel interface. May return Nothing.

Method	Description
SetObject(oObjIObj As IObject)	Sets the object to be deleted.

16.2. IRefObject

The properties of the *IRefObject* interface are shown below for your reference.

Property	Description
RefClass As String	Required property. Sets/returns the class of the reference object.
RefClassSetFlag As Boolean	Indicates whether RefClass has been set.
RefUID As String	Required property. Sets/returns the unique identifier for the reference object.
RefUIDSetFlag As Boolean	Indicates whether RefUID has been specified.
RefName As String	Optional property. Sets/returns the name of the reference object.
RefNameSetFlag	Indicates whether RefName has been specified.

Property	Description
As Boolean	
IObject As IObject	Returns the implied IObject interface.

16.3. Implement GetObjectStatusInTool

The first thing we need to do is create a Sys7DAL object according to the class of the object being deleted. This is passed in as the sClassUID parameter…

```
Select Case sClassUID
    Case "CSController"
        entity = New Sys7Controller()
    Case "CSIOCard"
        entity = New Sys7IOCard()
    Case "CSSignalChannel"
        entity = New Sys7Channel()
    Case "CSControlSysIO"
        entity = New Sys7IOTag()
End Select
```

Now, just try to read the object by the Id passed in the ??? parameter. If successful, respond that the object has been moved (and, therefore, still exists). Otherwise, respond that the object has been deleted…

```
If Not entity Is Nothing Then
    If entity.ReadById(m_db, sObjUID) Then
        ' ReadById successful so this object must still exist
        Return ObjectStatusInTool.Moved
    Else
        ' Did not find it, respond that it is deleted
        Return ObjectStatusInTool.Deleted
    End If
End If
```

The code for the entire method is as follows…

```
    Public Function GetObjectStatusInTool(ByVal sObjUID As String,
ByVal sClassUID As String) As SchemaCompInterfaces.ObjectStatusInTool
Implements SchemaCompInterfaces.IEFAdapter.GetObjectStatusInTool
        Dim entity As Sys7Entity = Nothing

        Select Case sClassUID
            Case "CSController"
                entity = New Sys7Controller()
            Case "CSIOCard"
                entity = New Sys7IOCard()
            Case "CSSignalChannel"
                entity = New Sys7Channel()
            Case "CSControlSysIO"
                entity = New Sys7IOTag()
        End Select

        If Not entity Is Nothing Then
            If entity.ReadById(m_db, sObjUID) Then
                ' ReadById successful so this object must still exist
                Return ObjectStatusInTool.Moved
            Else
                ' Did not find it, respond that it is deleted
```

```
            Return ObjectStatusInTool.Deleted
        End If
    End If
End Function
```

16.4. Implement GetRelationshipStatusInTool

In our application, there is actually no possibility that a relationship could be moved. In an application like SmartPlant P&ID, that could happen is the user just cuts some graphics from one drawing and pastes it into another. Since we don't have that possibility, we will simply respond that the rel has been deleted. Remember, this will only be called when a rel has been found missing from the previous publish.

```
    Public Function GetRelationshipStatusInTool(ByVal sRelUID As
String, ByVal sClassUID As String) As ObjectStatusInTool Implements
GetRelationshipStatusInTool
        Return ObjectStatusInTool.Deleted
    End Function
```

16.5. Process DeleteInstruction in RetrieveDocument

What we need to do in response to delete instructions is just get all the DeleteInstructions from the instructions (or tombstones) container and loop through them deleting the internal objects.

Let's add this functionality into a helper function called from RetrieveDocument…

```
RetrieveDeleteInstructions(oDocTombstonesIContainer)
```

And now in the helper function, get a collection of the delete instructions…

```
        Dim deleteInstructions As IObjectCollection = New
ObjectCollection()

instructionsContainer.IContainerQuery.GetInstancesForInterfaceDef("IDe
leteInstruction", deleteInstructions)
```

Loop through the collection of results…

```
        For Each deleteInstruction As IDeleteInstruction In
deleteInstructions
```

Next, using the RelClass on the IRefObject in the DeleteInstruction, instance an appropriate object…

```
            Dim entity As Sys7Entity = Nothing
            Select Case deleteInstruction.IRefObject.RefClass
                Case "CSController"
                    entity = New Sys7Controller()
                Case "CSIOCard"
                    entity = New Sys7IOCard()
                Case "CSSignalChannel"
                    entity = New Sys7Channel()
                Case "CSControlSysIO"
                    entity = New Sys7IOTag()
            End Select
```

Attempt to read the object using the SameAsId. Remember, the DeleteInstructions are generated during publish using the pubishing tool's UID. This is our SameAsId tucked away when we correlated the object…

```
                If entity.ReadBySameAs(m_db,
deleteInstruction.IRefObject.RefUID) Then
```

If we found the object delete it. This is done simply by setting the object cache state to Deleted and calling the Write function. The DAL code will take care of deleting it from the database…

```
entity.CacheState = CacheStates.Deleted
entity.Write(m_db)
```

The complete code listing for the RetrieveDeleteInstructions function is as follows…

```
Private Sub RetrieveDeleteInstructions(ByVal instructionsContainer As
IContainer)
        ' Find all the delete instructions
        Dim deleteInstructions As IObjectCollection = New ObjectCollection()

instructionsContainer.IContainerQuery.GetInstancesForInterfaceDef("IDeleteInst
ruction", deleteInstructions)

        ' Loop through and process each one
        For Each deleteInstruction As IDeleteInstruction In deleteInstructions
            Dim entity As Sys7Entity = Nothing
            Select Case deleteInstruction.IRefObject.RefClass
                Case "CSController"
                    entity = New Sys7Controller()
                Case "CSIOCard"
                    entity = New Sys7IOCard()
                Case "CSSignalChannel"
                    entity = New Sys7Channel()
                Case "CSControlSysIO"
                    entity = New Sys7IOTag()
            End Select

            If Not entity Is Nothing Then
                If entity.ReadBySameAs(m_db,
deleteInstruction.IRefObject.RefUID) Then
                    entity.CacheState = CacheStates.Deleted
                    entity.Write(m_db)
                End If
            End If
        Next
    End Sub
```

16.6. Lab Exercise

Add Delete Instruction processing to your adapter.

You may use the Chapter 16 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

17. SameAs

In Chapter 11, we talked about Correlation; the process of deciding that an internal object is the same as an object from another application (or applications). As a reminder, the correlation algorithm is expected to be…

1. Honor incoming correlations – if another application says it's object is correlated to our, then honor that statement. Were going to ignore this step for now and come back to it in Chapter 17 – Same As.
2. If we have previously correlated an incoming object with an existing object, then this is an update.
3. Else, attempt to correlate the incoming object to an existing object by some set of property values (most commonly name). If this is successful, persist the correlation for later use.
4. Else, consider this a new objects to be created in the application database. Once created, however, it is correlated so the application should persist that for later use.

In Chapter 11, we implemented steps 2, 3, and 4 of that algorithm. In this chapter we will add the following to our adapter…

- Implement step 1 of the correlation algorithm to process incoming SameAs relations in RetrieveDocument.
- Publish SameAs relations for any object that has been correlated.

17.1. ISameAs

Correlation is communicated via SameAs relationships. Even though they are called relationships, they do not share anything in common with the normal relationships we've worked with so far; meaning.

Property	Description
ExternalUID As String	The UID of the external object correlated with the internal object. (Remember, when retrieving SameAs published by another tool, your application will be external and the publishing application will be the internal. When retrieving, of course, it is the opposite.)
LocalUID As String	The UID of the internal object relative to the publisher. (As stated above, reversed relative to the retriever.)
LocalUIDIObj As IObject	A reference to the local object in the data container.
SharedObjDefUID As String	The SharedObjDef scoping this SameAs.
SharedObjDefUIDIObj As IObject	A reference to the SharedObjDef in the schema container.

17.2. Retrieve SameAs

SameAs relationships are added to the data container upon publish so to retrieve them, we need to search that container. For starters though, let's abstract this out into a helper function called from RetrieveDocument. Since retrieving SameAs is the first step in the correlation algorithm we need to make sure this is the first step in RetrieveDocument…

```
RetrieveSameAs(oDocContIContainer)
RetrieveData(oDocContIContainer)
RetrieveDeleteInstructions(oDocTombstonesIContainer)
```

Now, in our RetrieveSameAs helper function, start by getting all the SameAs rels from the data container…

```
Dim sameAsRels As IObjectCollection = New ObjectCollection()
dataContainer.IContainerQuery.GetInstancesForInterfaceDef(
"ISameAs", sameAsRels)
```

Loop through each…

```
For Each sameAsRel As ISameAs In sameAsRels
```

Use the Local ClassDef to decide what type of object to create…

```
Dim sameAsEntity As Sys7Entity = Nothing

Select Case sameAsRel.LocalUIDIObj.ClassDefIObj.UID
    Case "CSController"
        sameAsEntity = New Sys7Controller()
    Case "CSIOCard"
        sameAsEntity = New Sys7IOCard()
    Case "CSSignalChannel"
        sameAsEntity = New Sys7Channel()
    Case "CSControlSysIO"
        sameAsEntity = New Sys7IOTag()
End Select
```

Attempt to find the object by searching for it using our internal ID…

```
If sameAsEntity.ReadById(m_db, sameAsRel.ExternalUID) Then
```

If found, record the SameAsId. This means that when we retrieve the data objects, there should already be correlation and an update will occur without having to correlation by name…

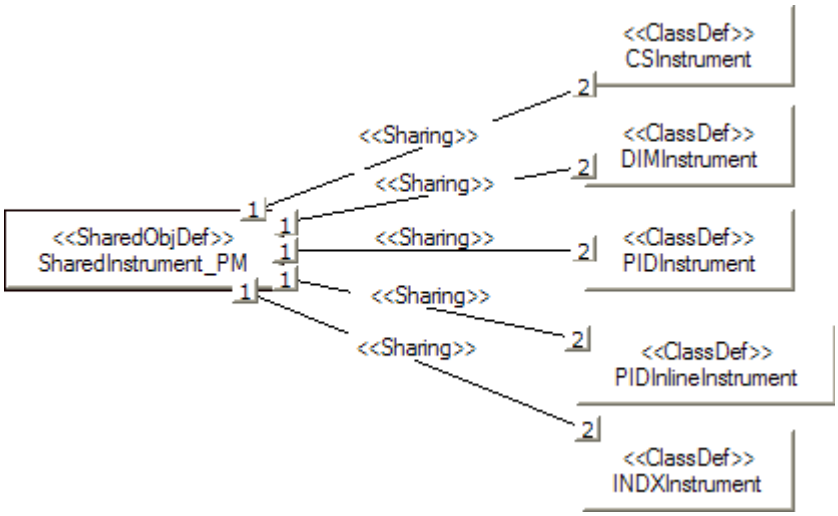
```
    sameAsEntity.SameAsId = sameAsRel.LocalUID
    sameAsEntity.Write(m_db)
```

The code for the RetrieveSameAs helper function is as follows…

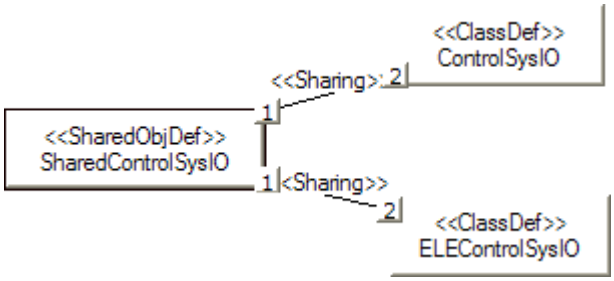
```
    Private Sub RetrieveSameAs(ByVal dataContainer As IContainer)
        ' Find all the SameAs rels
        Dim sameAsRels As IObjectCollection = New ObjectCollection()

dataContainer.IContainerQuery.GetInstancesForInterfaceDef("ISameAs",
sameAsRels)

        ' Loop through and process each one
        For Each sameAsRel As ISameAs In sameAsRels
            Dim sameAsEntity As Sys7Entity = Nothing

            Select Case sameAsRel.LocalUIDIObj.ClassDefIObj.UID
                Case "CSController"
                    sameAsEntity = New Sys7Controller()
                Case "CSIOCard"
                    sameAsEntity = New Sys7IOCard()
                Case "CSSignalChannel"
                    sameAsEntity = New Sys7Channel()
                Case "CSControlSysIO"
                    sameAsEntity = New Sys7IOTag()
            End Select

            If sameAsEntity.ReadById(m_db, sameAsRel.ExternalUID) Then
                sameAsEntity.SameAsId = sameAsRel.LocalUID
                sameAsEntity.Write(m_db)
            End If
        Next
    End Sub
```

17.3. Publish SameAs

There is one big requirement for publishing objects with the SameAs relationship; their ClassDefs must be part of a *Sharing* Relationship with the same *SharedObjDef* object in the schema. The following diagram shows the SharedObjDef for Instrument. (This diagram has actually been simplified.)



This shows that instruments coming from the Instrument Index (INDXInstrument), the Dimensional Datasheet (DIMInstrument), the P&ID (PIDInstrument & PIDInlineInstrument), and the control system (CSInstrument) can all be shared using the SharedInstrument_PM SharedObjDef. The following shows the SharedObjDef for IO tags…



Now, this might look like only objects coming from the control system and SmartPlant Electrical can be shared. However, SPI uses the same component schema (Control System Component) when it publishes the control system information. So, the control system application and SPI can still publish SameAs even though both use the same ClassDefs (e.g. ControlSysIO).

ISameAsHelper

Method	Description

Method	Description
CreateSameAs(oIContainer As IContainer, sUID As String, vLocalUID As Object, vExternalUID As Object, vSharedObjDefUID As Object	Creates a SameAs between the object with the local UID and the object with the external UID.

To publish the SameAs relationship, we just need to add a little code to each of the helper functions that publish our objects: PublishController, PublishIOCard, PublishChannel and PublishIOTag. We will only publish the SameAs if the object has a SameAsId (not NULL).

The following is the code to publish the SameAs for a controller. Note the use of the SameAsHelper, which simplifies the process…

```
        If controller.SameAsId.Length > 0 Then
            Dim sameAsHelper As ISameAsHelper = New Helper()
            sameAsHelper.CreateSameAs(dataContainer, controller.Id &
"-SameAs", controller.Id, controller.SameAsId, "SharedEquipment_PM")
        End If
```

For IO cards…

```
        If ioCard.SameAsId.Length > 0 Then
            Dim sameAsHelper As ISameAsHelper = New Helper()
            sameAsHelper.CreateSameAs(dataContainer, ioCard.Id & "-
SameAs", ioCard.Id, ioCard.SameAsId, "SharedIOCard_PM")
        End If
```

For channels…

```
        If channel.SameAsId.Length > 0 Then
            Dim sameAsHelper As ISameAsHelper = New Helper()
            sameAsHelper.CreateSameAs(dataContainer, channel.Id & "-
SameAs", channel.Id, channel.SameAsId, "SharedSignalChannel")
        End If
```

And for IO tags…

```
        If ioTag.SameAsId.Length > 0 Then
            Dim sameAsHelper As ISameAsHelper = New Helper()
            sameAsHelper.CreateSameAs(dataContainer, ioTag.Id & "-
SameAs", ioTag.Id, ioTag.SameAsId, "SharedControlSysIO")
        End If
```

17.4. Lab Exercise

Add SameAs retrieve and publish to your adapter.

You may use the Chapter 17 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

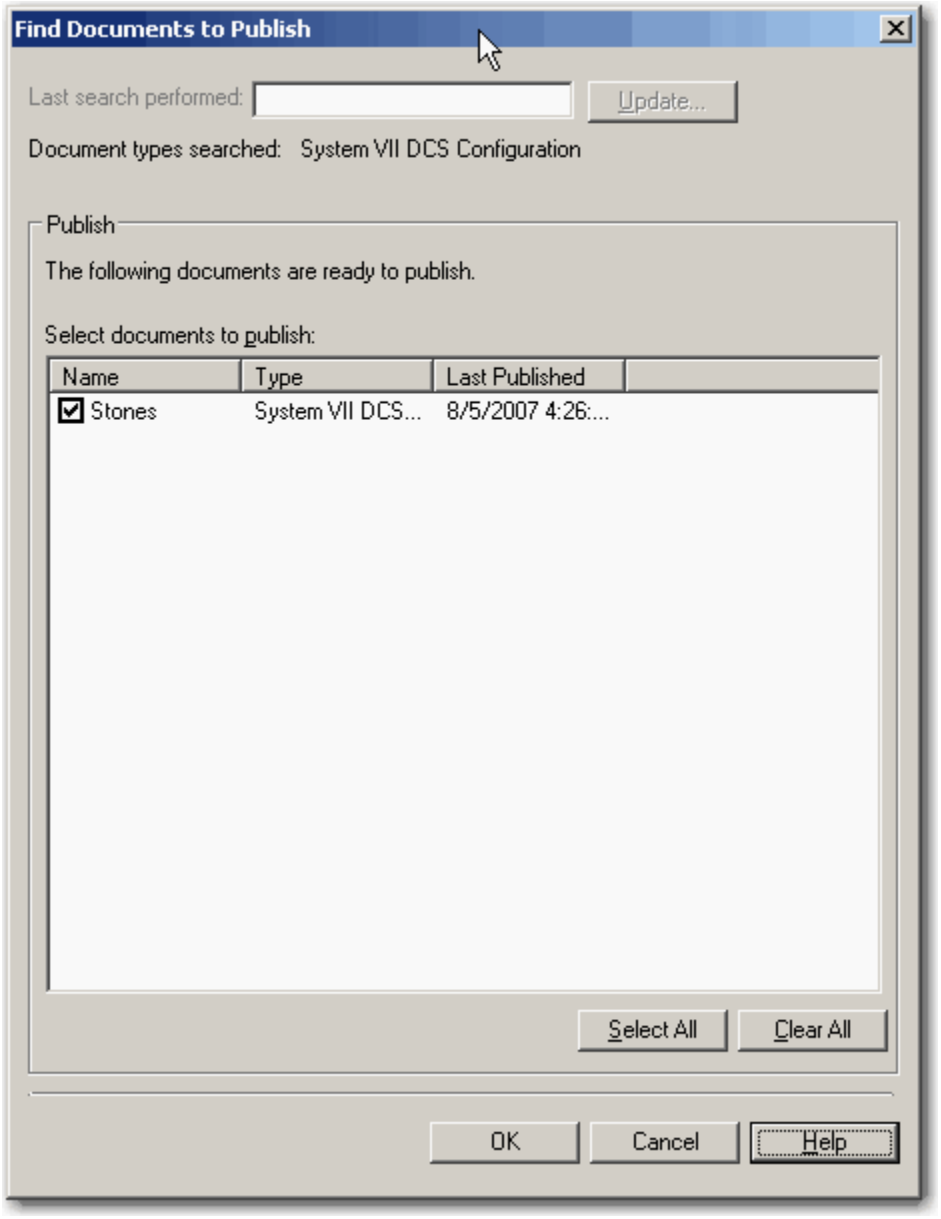
18. Find Documents to Publish

The Find Documents to Publish command is another major SmartPlant menu command. However, its implementation is optional. All of the SmartPlant 2D and 3D engineering applications have implemented this command.

There are two benefits to this command…

1. An application may contain hundreds or even thousands of documents/drawings, which would make it quite a task to keep up with which ones need to be published and which ones do not. This command automates that process; valuable to those times when all the deliverables need to be published to meet a given milestone.
2. When the document itself is deleted (or the scope of the publish is deleted), you can no longer publish that scope. That means there is no opportunity for SPF to discover that delete and generate instructions. In our example application, the scope of the publish is a controller. If we delete a controller, we can no longer publish it and SPF won't discover its deletion. The Find Documents to Publish command implements an algorithm to discover deleted documents/scopes and generate delete instructions appropriately.

Again, the SmartPlant Client (EF Common UI) provides the GUI for the command. The actual GUI you see may be different because the behavior is a bit different depending on the results.



In the snapshot above, SmartPlant (by querying the adapter) has discovered one document that should be published. It is also possible to discover new documents (those that have never been published) and documents that have been deleted. The GUI will look slightly different depending on what is found.

Implementing Find Documents to Publish in your application simply means there is some algorithm or heuristic in place to determine that application documents need to be published (instead of allowing a user to publish documents at will).

In effect, this is the example application's way of setting a document's state to denote it needs to be published (because the contents of the document have changed after retrieving SPI data and updating the application database). In our example application, we'll determine this via the IsDocumentDirty property on the Controller object. If data

is added to the controller, this flag should be set to True indicating that it has changed since last publish.

Note: One common methodology with applications that have the concept of a document is to have a Date/Time stamp associated with its documents.

For example, there could be a *LastModified* and a *LastPublished* value for each document. If the LastModified value is later than the LastPublished value, it would be appropriate to publish the document again.

The steps we need to follow to support Find Documents to Publish are as follows…

- Implement a menu handler in the application for the Find Documents to Publish command on the SmartPlant menu.
- Implement the FindDocsToPublish method on the adapter.
- Implement the DocumentExistsInTool method on the adapter.
- Implement the PublishConfirm method on the adapter.
- In the SupportsFeature adapter method, respond True when asked if we support Find Documents to Publish.

18.1. Application Menu Handler

The code behind the Find Documents to Publish menu command looks very similar to Publish or Retrieve. The only real difference being that, after connecting to SPF, you will call FindDocsToPublish rather than ShowPublishDialog or ShowRetrieveDialog.

There should be nothing new to you here so the entirety of the function is as follows…

```
    Private Sub SmartPlantFindDocsMenu_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
_smartPlantFindDocsMenu.Click
        Try
            ' Need the registration information out of the selected
plant
            Dim plant As Sys7Plant = GetSelectedPlant()
            If plant Is Nothing Then Throw New Exception("Error
getting selected plant.")

            ' Create an instance of the SmartPlant Common UI
            Dim commonUIApplication As IEFCommonUIApplication
            commonUIApplication =
CreateObject("EFCommonUI.Application")
            If commonUIApplication Is Nothing Then Throw New
Exception("Error starting SPF Common UI.")

            ' Set up the tool paramters
            Dim toolParameters As IEFToolParameters
            toolParameters = New EFToolParameters()
            If toolParameters Is Nothing Then Throw New
Exception("Error instancing tool parameters.")

            ' Send needed information over to the adapter through the
tool parameters
            toolParameters.Add("Database", _db, "Database")
            toolParameters.Add("Plant", plant, "Plant")

            ' Create an instance of the tool docs (even though we
won't be using it right now
            Dim toolDocs As IEFToolDocs
            toolDocs = CreateObject("Sys7Adapter.ToolDocs")
            If toolDocs Is Nothing Then Throw New Exception("Error
instancing tool docs.")

            ' Connect to SPF
            Dim result As Integer =
commonUIApplication.Connect(plant.SpfUrl, plant.SpfPlant, "", "",
"Sys7DS", "Sys7Adapter.Adapter", toolParameters, toolDocs)
            If result < 0 Then Throw New Exception("Error connecting
to SPF. [" & commonUIApplication.LastErrorMessage & "]")

            ' Start the find docs to publish process. The SmartPlant
client takes over from here and will make use of the adapter
            ' during the process.
            result = commonUIApplication.FindDocsToPublish()
            If result < 0 Then Throw New Exception("Error during
Publish. [" & commonUIApplication.LastErrorMessage & "]")
```

```
            ' Reinitialize tree
            InitializeTree()

        Catch exp As Exception
            MessageBox.Show("An error occurred during the publish
process. " & exp.Message, "Publish", MessageBoxButtons.OK)
        End Try
    End Sub
```

18.2. FindDocsToPublish Method

To find documents that need to be publish (because they have changed since last publish), the SmartPlant client will call the FindDocsToPublish API on the adapter. The application should then consult the application data store to discover documents that have changed.

In our example application, document is one-to-one with controller. So, we'll be looking for controllers that have an IsDocumentDirty property set to True. Any documents that should be published must be instanced in the oDocsToPublish container.

Note: If your application keeps track of documents that have never been published before (they are *new* documents), then create the document in the *oNewDocsToPublishIContainer* container passed into *FindDocsToPublish* (instead of *oDocsToPublishContainer*).

These documents should be created using the Document ClassDef in the same way we created the active document in the application before calling ShowPublishDialog. That means we need to ability to create an instance of a document in both the application and the adapter. That suggests that the code should be shared in some common layer. For now, just copy the CreateDocument method into the adapter.

To find all the controllers that need to be published, we need to get all the controllers. However, you need to remember that we are connected to a given plant so the list of controllers should be scoped by that plant. There is a method on the Plant object in the DAL to help with this: ReadChildControllers. Calling this from any Plant will return a collection of controllers…

```
        Dim controllers As ICollection =
m_plant.ReadChildControllers(m_db)
        If controllers Is Nothing Then Throw New Exception("Error
reading controllers in plant.")
```

For each controller which has True for IsDocumentDirty…

```
    For Each controller As Sys7Controller In controllers
            If controller.IsDocumentDirty Then
```

Create that document in the oDocsToPublish container…

```
                CreateDocument(oDocsToPublishIContainer, controller.Id
& "-Document", controller.Name, "", "Control system configuration for
" & controller.Name, "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}")
```

The complete listing of this function is as follows…

```
    Public Sub FindDocsToPublish(ByRef oDocsToPublishIContainer As
SchemaCompInterfaces.IContainer, ByRef oNewDocsToPublishIContainer As
SchemaCompInterfaces.IContainer) Implements
SchemaCompInterfaces.IEFAdapter.FindDocsToPublish
        ' Get all the controllers in this plant
        Dim controllers As ICollection =
m_plant.ReadChildControllers(m_db)
        If controllers Is Nothing Then Throw New Exception("Error
reading controllers in plant.")

        ' Loop through testing the dirty flag on each one. Create
document objects for all dirty
        For Each controller As Sys7Controller In controllers
            If controller.IsDocumentDirty Then
                CreateDocument(oDocsToPublishIContainer, controller.Id
& "-Document", controller.Name, "", "Control system configuration for
" & controller.Name, "{64CBA09B-50D3-4FF2-8EE3-7725871F8728}")
            End If
        Next
    End Sub
```

18.3. DocumentExistsInTool Method

During the Find Document to Publish process, SmartPlant will also attempt to discover any deleted documents/scopes. It does this by inquiring about any documents that have been published in the past. SmartPlant Foundation will always have the complete Publish history of any application. For each document an application has published, the SmartPlant Client will inquire as to its status by calling the DocumentExistsInTool method on the adapter. The adapter should look in the application data store and respond True if it still exists and False if it does not.

Since our Document is 1:1 with a Controller, we will just need to covert the document UID to a controller UID and determine if that controller still exists. We've implemented a function ControllerIdFromDocumentId to help with this…

```
        Dim controllerId As String = ControllerIdFromDocumentId(
oDocumentIObj.UID)
```

Try to read the controller by Id and return True if successful and False if not…

```
        Dim controller As Sys7Controller = New Sys7Controller()
        If controller.ReadById(m_db, controllerId) Then
            Return True
        Else
            Return False
        End If
```

The entire listing for this function is as follows…

```
    Public Function DocumentExistsInTool(ByRef oDocumentIObj As
IObject) As Boolean Implements IEFAdapter.DocumentExistsInTool
        ' Go get the controller associated with this document
        Dim controllerId As String = ControllerIdFromDocumentId(
oDocumentIObj.UID)
        Dim controller As Sys7Controller = New Sys7Controller()
        If controller.ReadById(m_db, controllerId) Then
            Return True
        Else
            Return False
        End If
    End Function
```

18.4. PublishConfirm Method

The PublishConfirm method is where the adapter should clear an IsDirty flag or set the LastPublish date-time. Whatever the implementation, this is the place to do it. The PublishDocument method is not appropriate for this because there are steps after that method that could cause a publish to fail. The data may fail validation, communication with SPF could be interrupted, or an internal error in SPF could all cause a publish to fail subsequent to the completion of PublishDocument.

If a publish is successful, however, the SmartPlant Client will call the PublishConfirm method on the adapter. For our example adapter, we just need to set the IsDocumentDirty flag on the Controller to False. First, convert the document Id to a controller Id and get the controller from the database…

```
        Dim controllerId As String =
ControllerIdFromDocumentId(oDocumentIObj.UID)
        Dim controller As Sys7Controller = New Sys7Controller()
        If Not controller.ReadById(m_db, controllerId) Then Throw New
Exception("Error finding controller from document.")
```

Now, just set the IsDocumentDirty property to False and write it back to the database…

```
        controller.IsDocumentDirty = False
        controller.Write(m_db)
```

The complete listing of this function is as follows…

```
    Public Sub PublishConfirm(ByRef oDocumentIObj As IObject)
Implements IEFAdapter.PublishConfirm
        ' Go get the controller associated with this document
        Dim controllerId As String =
ControllerIdFromDocumentId(oDocumentIObj.UID)
        Dim controller As Sys7Controller = New Sys7Controller()
        If Not controller.ReadById(m_db, controllerId) Then Throw New
Exception("Error finding controller from document.")

        controller.IsDocumentDirty = False
        controller.Write(m_db)
    End Sub
```

18.5. Change SupportsFeature Method

As you recall from Chapter 9, we implemented the *SupportsFeature* method and set it to always return False. We now need to modify this routine to advise The Engineering Framework that we now support the *FindDocsToPublish* feature:

```
Select Case Feature
            Case ToolFeatures.FindToPublish
                SupportsFeature = True
            Case Else
                SupportsFeature = False
        End Select
```

18.6. Lab Exercise

Write the code to implement FindDocsToPublish.

You may use the Chapter 18 – Lab 1 folder as a starting point.

The completed lab exercise is available in the *Solution* project folder.

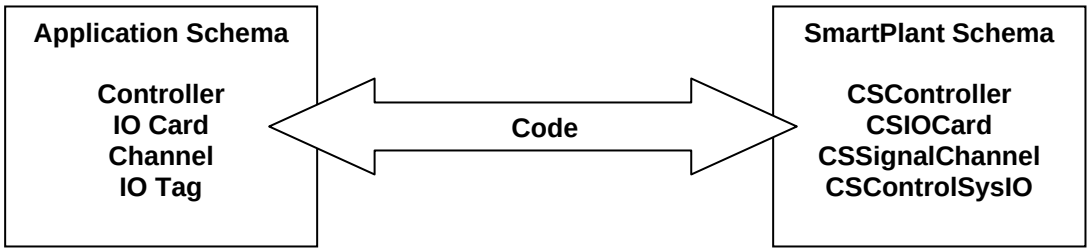
19. Tool Schema Modeling and Mapping

By now, you may have noticed that there are several fundamental flaws in our solution. First, how many times have we implemented a Select statement that creates an internal Class based on a SmartPlant Schema ClassDef? Of course, we can factor this code out and write it only once, but we are still hard-coded to a given ClassDef.

Secondly, you might have noticed that we have only published or retrieved IObject properties so far: UID, Name, and Description. To publish or retrieve more properties, you can see that we would need to add more code to deal with the additional InterfaceDefs and PropertyDefs. Hard-coding this may be ok if you're working with an application that has a fixed data model and you don't foresee much change. However, if wanted to add an additional property to the example application, in addition to adding that property to the database, the DAL, and the application GUI, we'd also have to add code to support it in pubish & retrieve.

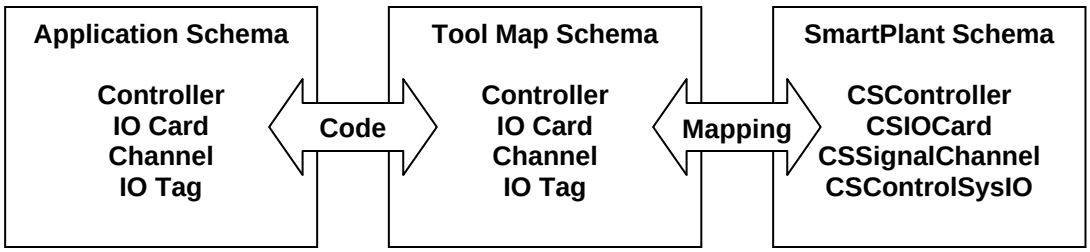
SmartPlant Integration offers a way to externalize these mapping and allow you to implement generic code for publish and retrieve. Of course, this does not make your application data model dynamic in itself. But it does reduce the amount of code that must be changed when that data model changes. It also unbinds your code from the SmartPlant Schema. If the Schema changes, you can change the externalized mapping and not the code.

19.1. How does it work?



Whether you know it or know, your application has a schema. This may be represented by the tables and columns in a database or it may be represented by a complex Business Layer within your application. Either way, there is an application schema. In our example application, we have both a database schema and a data access layer with business objects of controller, IO card, channel, and IO tag.

We've also seen that these same objects are represented in the SmartPlant Schema as ClassDefs CSController, CSIOCard, CSSignalChannel, and CSControlSysIO. Until now, though explicit code has provided the mapping between the two schemas.



As stated earlier, SmartPlant Mapping externalizes the mapping so that your code is not tightly bound to the SmartPlant Schema. To externalize the application schema, a Tool Map Schema is created. The Tool Map Schema is created, maintained, and delivered by each application as an XML file separate from the SmartPlant Schema. For the most part, the Tool Map Schema may be identical to the Application Schema. This is not required, but many times makes the most sense.

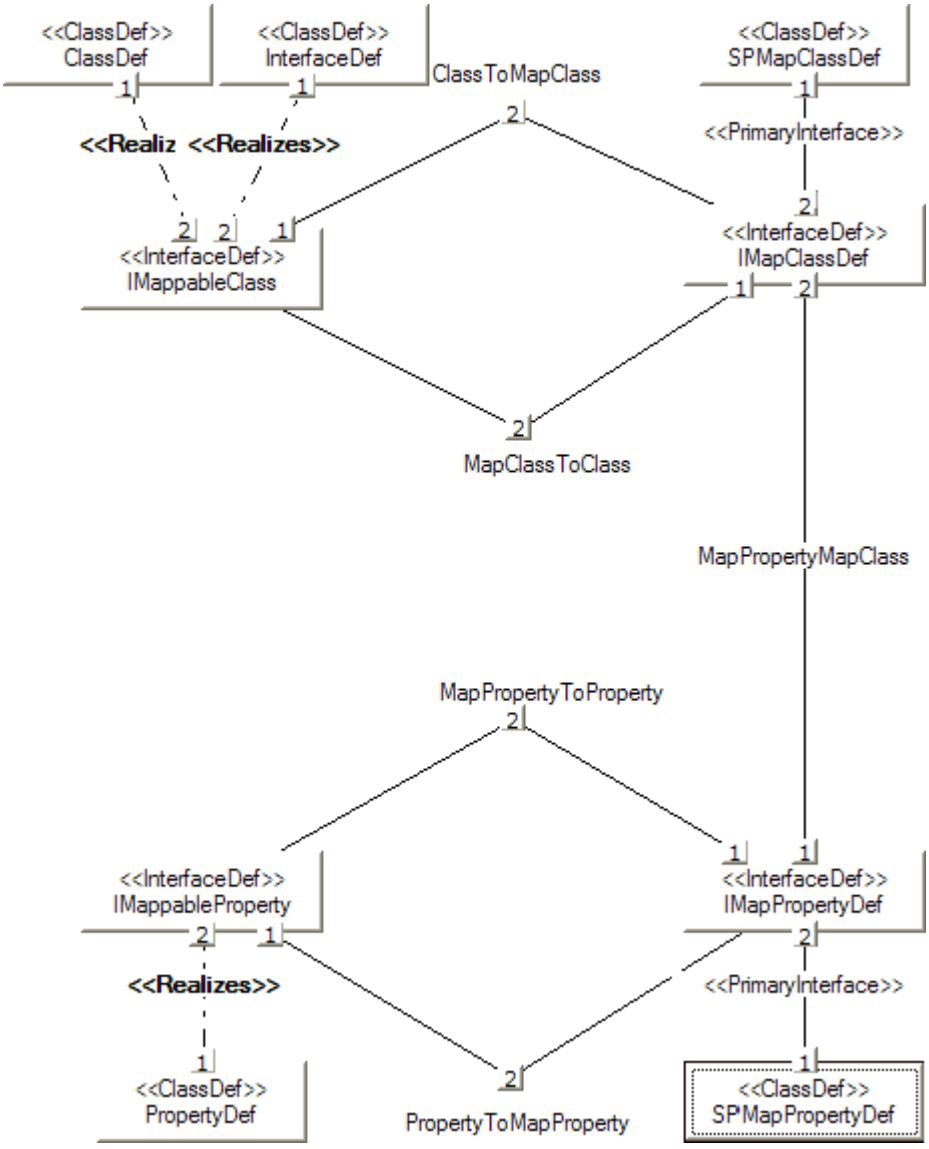
In our example, we'll model the Tool Map Schema to match the DAL as closely as possible. This will make the code that maps between the two trivial.

Now, the mapping between the Tool Map Schema and the SmartPlant Schema is probably not trivial because your application was built independently of the SmartPlant Schema and will not match. The mapping between the two is created using the Schema Editor and stored within the separate Tool Map Schema file.

In this chapter, we're going to change our retrieve code to a generic algorithm using SmartPlant Mapping.

19.2. Tool Map Schema

The following diagram illustrates the way that mapping is modeled within SmartPlant…



Note: Some of the interfaces and relationships involved have been left out of this diagram to allow you to concentrate on the classes and rels applicable to mapping.

On the left, you see ClassDef, InterfaceDef, and PropertyDef from the SmartPlant Schema. On the right, you see two new classes: SPMapClassDef and SPMapPropertyDef. These two classes are what you use to create the Tool Schema.

Notice that both ClassDef and InterfaceDef realize an Interface called IMappableClass. This is because SmartPlant Integration allows you to map to either ClassDefs or InterfaceDefs. However, even though it is possible to map to ClassDefs, Intergraph highly recommends that

you map (at least for retrieving) to InterfaceDefs. The reason is because ClassDefs can be application specific. For instance, if you want to retrieve instruments from both SPP&ID and SPI, you might have to map to several ClassDefs: PIDInstrument, PIDInlineInstrument, INDXInstrument, DIMInstrument, etc. And for each application added, you'd have to do more mapping. However, if you were to map to something like IInstrumentOcc, you could retrieve instruments from any applications.

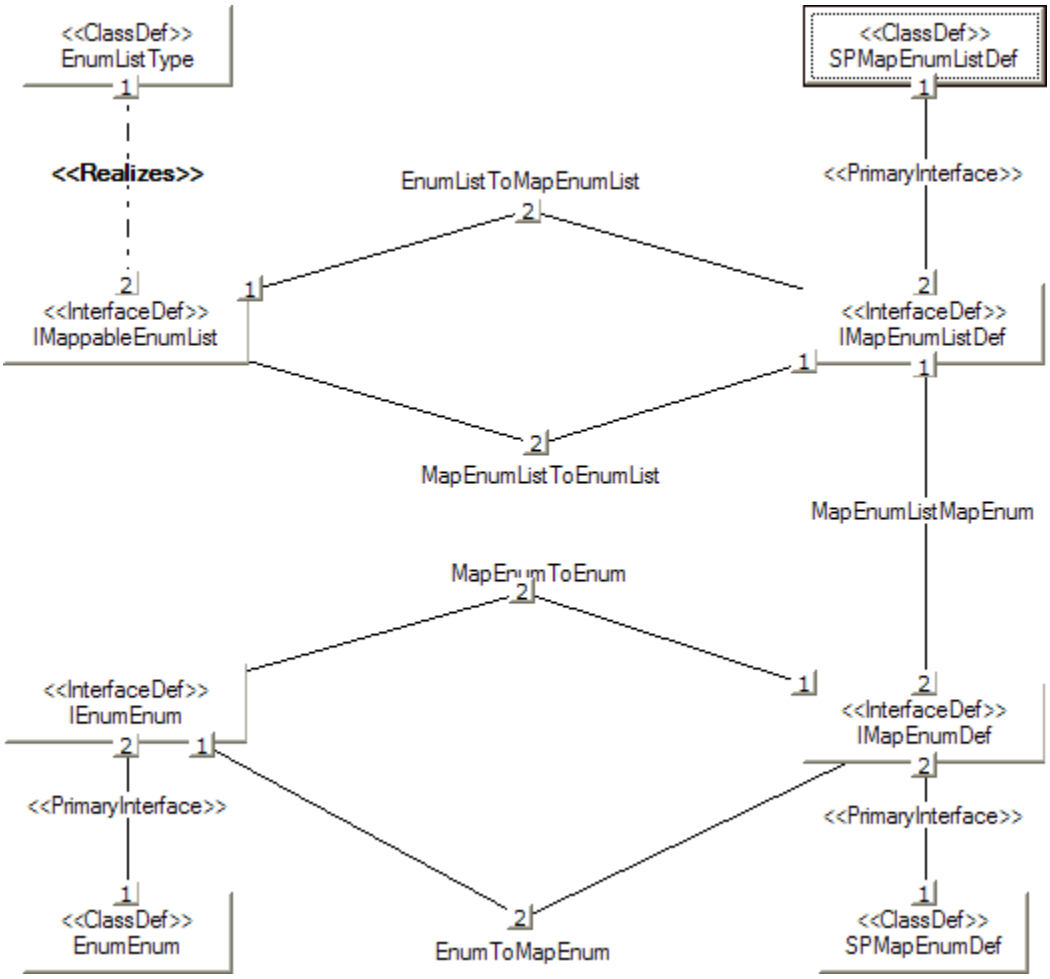
In the diagram above, you see that ClassDefs/InterfaceDefs are mapped to SPMapClassDefs with two, directional RelDefs: ClassToMapClass and MapClassToClass. It's modeled this way to allow the Publish and Retrieve maps to differ. If you wanted to navigate the mapping from an InterfaceDef to an SPMapClass, you would choose the ClassToMapClass rel. If you wanted to navigate the mapping between an SPMapClass and the mapped InterfaceDef, you would choose the MapClassToClass rel.

Notice that a similar model exists to map PropertyDefs to SPMapPropertyDefs. Two, directional RelDefs provide the Publish and Retrieve mapping: MapPropertyToProperty and PropertyToMapProperty.

Also notice that the RelDef between SPMapClassDef and SPMapPropertyDefs is called MapPropertyMapClass.

19.3. Mapping of Enumerated Lists

It is a given that the enumerated values in your application will be different than those defined in the SmartPlant Schema. SmartPlant also offers a mapping solution for this…



EnumListTypes are mapped to a Tool Schema class of SPMapEnumList through two RelDefs: EnumListToMapEnumList and MapEnumListToEnumList.

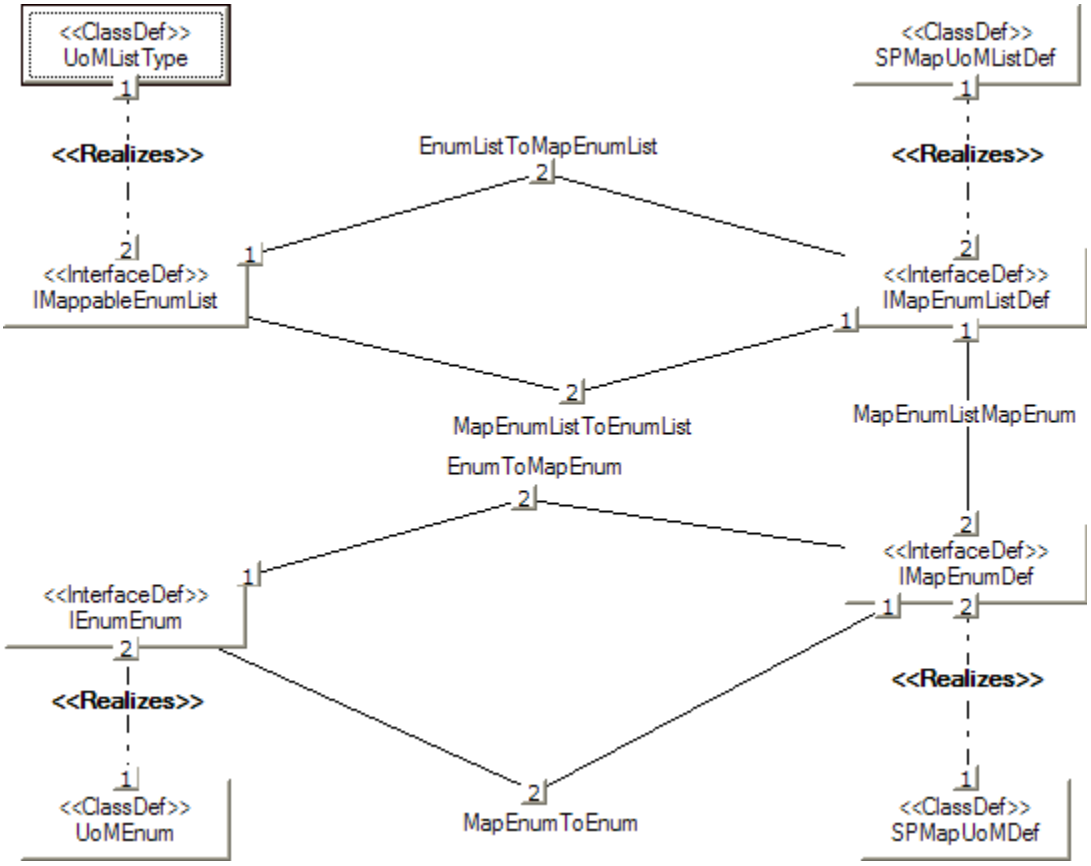
An SPMapEnumList contains SPMapEnumDefs using the RelDef MapEnumListMapEnum.

And EnumEnums are mapped to a Tool Schema class of SPMapEnumDef through to RelDefs: EnumToMapEnum and MapEnumToEnum.

By navigating these relationships in the schema, an adapter can retrieve enumerated values and translated them into the values used internally.

19.4. Mapping of UomLists

SmartPlant also offers mapping of Uoms and, due to the fact that UomListTypes and UomEnums share common interfaces with EnumListTypes and EnumEnums, the implementation of that mapping is very similar.



UomListTypes are mapped to a Tool Schema class of SPMapUomListDef through two RelDefs: EnumListToMapEnumList and MapEnumListToEnumList.

An SPMapUomListDef contains SPMapUomDefs using the RelDef MapEnumListMapEnum.

And UomEnums are mapped to a Tool Schema class of SPMapUomDef through to RelDefs: EnumToMapEnum and MapEnumToEnum.

By navigating these relationships in the schema, an adapter can retrieve UoM values and translated them into the units of measure used internally.

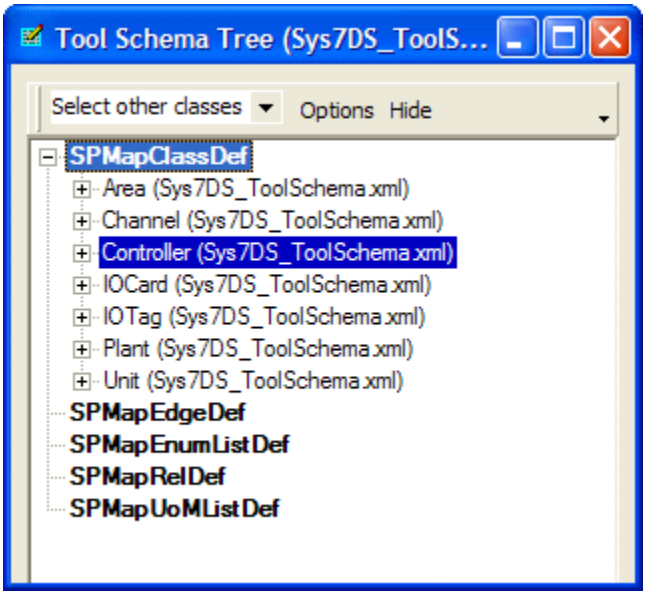
19.5. Implementing SmartPlant Mapping

The steps required to implement SmartPlant Mapping are as follows…

- Create a Tool Schema and map into the SmartPlant Schema.
- Change the SupportsFeature method in the adapter to respond true to the Mapping feature.
- Implement the GetMapSchemaFile method in the adapter to tell SmartPlant where your Tool Map Schema is.
- Modify your publish code in the adapter to create classes, interfaces, and set property values according to the mapping.
- Modify your retrieve code in the adapter to create classes and property values according to the mapping.

19.5.1. Create A Tool Map Schema

Using the Schema Editor to map is covered in the Schema Editor and Modeling course and this will not be repeated here. You have been provided with a Tool Map Schema already prepared for you. Feel free to review that in detail using the Schema Editor.



19.6. SupportsFeature for Mapping

Since SmartPlant mapping is an optional feature, we need to inform the SmartPlant Client that we support it. Once again, add a new feature into the SupportsFeature method on the adapter.

```
    Public Function SupportsFeature(ByVal Feature As ToolFeatures,
Optional ByRef sDocTypeUID As String = Nothing) As Boolean Implements
IEFAdapter.SupportsFeature
        Select Case Feature
            Case ToolFeatures.FindToPublish
                SupportsFeature = True
            Case ToolFeatures.Mapping
                SupportsFeature = True
            Case Else
                SupportsFeature = False
        End Select
    End Function
```

19.6.1. Implement GetMapSchemaFile

The implementation of this method is trivial. We just need to return a string containing the full path to our Tool Map Schema file.

```
    Public Function GetMapSchemaFile(ByRef sDocTypeUID As String) As
String Implements IEFAdapter.GetMapSchemaFile
        GetMapSchemaFile = "D:\SP_Adapter\Sys7DS_ToolSchema.xml"
    End Function
```

Note that the document type is passed as an input to this method. This allows the adapter to present a different Tool Map Schema for each document type. Unless you have a good reason to do differently, it is simpler to just have a single Tool Map Schema.

19.6.2. Implement Publish Mapping

Due to time constraints, we will not implement publish mapping in our example adapter.

19.6.3. Implement Retrieve Mapping

The first thing we want to do is get rid of that Select statement hard-coding the mapping between the SmartPlant Schema ClassDefs and our application classes. We'll replace that with a helper function GetMapClassForObject…

```
            Dim mappedClassIObject As IObject = Nothing
            Dim mapClassIObject As IObject =
GetMapClassForObject(oIObject, mappedClassIObject)
```

We next factor out the correlation algorithm into a helper function called CorrelateOrCreate. Here's how it is called from RetrieveData…

```
                Dim sys7Object As Sys7Entity = Nothing
                If Not mapClassIObject.Name = "Plant" Then
                    ' Call the normal correlation algorithm
                    sys7Object = CorrelateOrCreate(oIObject,
mapClassIObject)
                Else
                    ' Plants are correlated manually during the register
process
                    sys7Object = m_plant
                    sys7Object.SameAsId = oIObject.UID
                End If
```

Note that we're still treating plant special because it is manually correlated during register. The only thing different in the CorrelateOrCreate function is that we use the SPMapClassDef to instance a new object. Our DAL has a ClassFactory object that will create any internal class based on the map class name…

```
        Dim sys7Object As Sys7Entity =
classFactory.Create(mapClassIObject.Name)
```

The final change we will make to our RetrieveData method is to retrieve the object properties based on the mapping. We'll put this capability into a helper function called RetrieveObjectProperties. Here's how it is called from RetrieveData…

```
                RetrieveObjectProperties(sys7Object, oIObject,
mapClassIObject, mappedClassIObject)
```

The complete listing for our new RetrieveData method implementing SmartPlant Mapping is as follows…

```
Private Sub RetrieveData(ByVal dataContainer As IContainer)
        ' Set up some stuff we'll need later on
        Dim classFactory As Sys7ClassFactory = New Sys7ClassFactory()
        Dim pbsEntity As Sys7Entity = Nothing
        Dim parentRel As IRel = Nothing
        Dim relHelper As IRelHelper = New Helper

        For Each oIObject As IObject In dataContainer.ContainedObjects

            Dim mappedClassIObject As IObject = Nothing
            Dim mapClassIObject As IObject =
GetMapClassForObject(oIObject, mappedClassIObject)
            If Not mapClassIObject Is Nothing Then

                Dim sys7Object As Sys7Entity = Nothing
                If Not mapClassIObject.Name = "Plant" Then
                    ' Call the normal correlation algorithm
                    sys7Object = CorrelateOrCreate(oIObject,
mapClassIObject)
                Else
                    ' Plants are correlated manually during the register
process
```

```
                    sys7Object = m_plant
                    sys7Object.SameAsId = oIObject.UID
                End If

                ' Retrieve property values based on the mappings
                RetrieveObjectProperties(sys7Object, oIObject,
mapClassIObject, mappedClassIObject)

                ' Get the ParentId
                Dim parentId As String = GetParentId(oIObject)
                If parentId.Length > 0 Then
                    sys7Object.ParentId = parentId
                    sys7Object.ResolveParentId(m_db)
                End If

                ' That's it. Write the changes.
                sys7Object.Write(m_db)
            Else
                ' Could find no mappings to this object's classdef nor
any interface
            End If
        Next
    End Sub
```

19.6.4. GetMapClassForObject Function

The first thing the helper function needs to do is to see if the object's ClassDef is mapped to one of our SPMapClassDefs…

```
        Dim classToMapClassIRel As IRel =
relHelper.GetRelForDefUID(objectIObject.ClassDefIObj.End1IRelCollection,
"ClassToMapClass")
```

This would only work if the mapping is to ClassDefs. Remember that we actually recommend mapping to InterfaceDefs. If we fail to find a mapping to the ClassDefs, we need to check the mapping on each of the object's interfaces. Notice that we are not going to follow the ClassDefs Realizes to InterfaceDefs. That is because we do not want to find a mapping if the object does not have this Interface. So, we need to get a list of the Interfaces instanced on the object…

```
            For Each oIInterface As IInterface In
objectIClassDefComp.InterfaceCollection(False)
```

Now, see if the InterfaceDef is mapped into our Tool Map Schema…

```
                classToMapClassIRel =
relHelper.GetRelForDefUID(oIInterface.InterfaceDefIObj.End1IRelCollection
, "ClassToMapClass")
```

As soon as we find a mapping, return both the mapped ClassDef/InterfaceDef and the SPMapClassDef to the caller.

The complete listing for the GetMapClassForObject method is as follows…

```
Private Function GetMapClassForObject(ByVal objectIObject As IObject,
ByVal mappedClassIObject As IObject) As IObject
        ' We'll need a rel helper
        Dim relHelper As IRelHelper = New Helper()

        ' Try to get a rel over to the map class
        Dim classToMapClassIRel As IRel =
relHelper.GetRelForDefUID(objectIObject.ClassDefIObj.End1IRelCollection,
"ClassToMapClass")
        If Not classToMapClassIRel Is Nothing Then
            ' This only works if mapped by class
            mappedClassIobject = classToMapClassIRel.UID1IObj
            Return classToMapClassIRel.UID2IObj
        Else
            ' No class mappings, look for interface mappings
            Dim objectIClassDefComp As IClassDefComponent = objectIObject

            For Each oIInterface As IInterface In
objectIClassDefComp.InterfaceCollection(False)
                classToMapClassIRel =
relHelper.GetRelForDefUID(oIInterface.InterfaceDefIObj.End1IRelCollection
, "ClassToMapClass")
                If Not classToMapClassIRel Is Nothing Then
                    ' Found a mapping to an interface
                    mappedClassIObject = classToMapClassIRel.UID1IObj
                    Return classToMapClassIRel.UID2IObj
                End If
            Next
        End If
        Return Nothing
    End Function
```

19.6.5. CorrelateOrCreate Function

The complete listing for the CorrelateOrCreate method is as follows…

```
Private Function CorrelateOrCreate(ByVal objectIObject As IObject, ByVal
mapClassIObject As IObject) As Sys7Entity
        ' We'll need a class factory
        Dim classFactory As Sys7ClassFactory = New Sys7ClassFactory()

        ' Ask the class factory to create an object based on the map
class
        Dim sys7Object As Sys7Entity =
classFactory.Create(mapClassIObject.Name)
        If sys7Object Is Nothing Then Throw New Exception("Error creating
object from map class.")
```

```
        ' Attempt to read by same as id
        If Not sys7Object.ReadBySameAs(m_db, objectIObject.UID) Then
            ' Attempt to correlate by name
            If sys7Object.ReadByName(m_db, objectIObject.Name) Then
                ' We just correlated by name, set the same as id
                sys7Object.SameAsId = objectIObject.UID
            Else
                ' Else it is a create. Still, it's now correlated so set
the same as id
                sys7Object.SameAsId = objectIObject.UID
            End If
        End If

        Return sys7Object
    End Function
```

19.6.6. RetrieveObjectProperties Function

This is the most complicated piece of code you will encounter in this course. First, we already have the SPMapClassDef, so follow the MapClassMapProperties rel to the collection of SPMapPropertyDefs…

```
        Dim mapProperties As IRelCollection =
relHelper.GetRelsForDefUID(mapClassIObject.End1IRelCollection,
"MapClassMapProperties")
```

For each of those SPMapPropertyDefs, follow the PropertyToMapProperty rel to the mapped PropertyDef…

```
                Dim propertyMapPropertyIRel As IRel =
relHelper.GetRelForDefUID(mapPropertyIObject.End2IRelCollection,
"PropertyToMapProperty")
```

To access this property on the incoming object, we'll have to know what InterfaceDef exposes it. (Call to GetInterface followed by a call to GetProperty.) Follow the Exposes rel to the InterfaceDef…

```
                    Dim exposesIRel As IRel =
relHelper.GetRelForDefUID(propertyDefIObject.End2IRelCollection,
"Exposes")

Now do a GetInterface and then GetProperty (on the incoming object)…
                    Dim oIInterface As IInterface =
objectIClassDefComp.GetInterface(interfaceDefIObject.UID, False)
                        oIProperty =
oIInterface.GetProperty(propertyDefIObject.UID, False)
```

Our DAL classes all have functions that allow us to access the properties using the property names. These functions are GetValue and SetValue, which provides access to the properties

through the SPMapPropertyDef name. Using this, we can now set the property on the internal object from the property value on the incoming object…

```
toolObject.SetValue(mapPropertyIObject.Name, oIProperty.Value)
```

Note: For simplicity, this example treats all properties as String types. For other types, you would want to switch on the PropertyType and deal with them accordingly. EnumListTypes, for example, would probably warrant a helper function on their own to translate from the EnumEnums to the SPMapEnumDefs.

The complete listing for the RetrieveObjectProperties helper function is as follows…

```
Private Sub RetrieveObjectProperties(ByVal toolObject As Sys7Entity,
ByVal objectIObject As IObject, ByVal mapClassIObject As IObject, ByVal
mappedClassIObject As IObject)
        ' Need a rel helper
        Dim relHelper As IRelHelper = New Helper

        ' Get a list of all the properties that are mapped
        Dim mapProperties As IRelCollection =
relHelper.GetRelsForDefUID(mapClassIObject.End1IRelCollection,
"MapClassMapProperties")
        If Not mapProperties Is Nothing Then

            ' Loop through each mapped property
            For Each mapPropertyIRel As IRel In mapProperties
                Dim mapPropertyIObject As IObject =
mapPropertyIRel.UID2IObj

                ' We have the SPMapProperty, follow the relation to the
schema property
                Dim propertyMapPropertyIRel As IRel =
relHelper.GetRelForDefUID(mapPropertyIObject.End2IRelCollection,
"PropertyToMapProperty")
                If Not propertyMapPropertyIRel Is Nothing Then

                    ' Have the rel, get the property def
                    Dim propertyDefIObject As IObject =
propertyMapPropertyIRel.UID1IObj

                    ' Now follow the exposes rel (to the InterfaceDef)
                    Dim exposesIRel As IRel =
relHelper.GetRelForDefUID(propertyDefIObject.End2IRelCollection,
"Exposes")

                    ' Get the InterfaceDef
                    Dim interfaceDefIObject As IObject =
exposesIRel.UID1IObj

                    Dim objectIClassDefComp As IClassDefComponent =
objectIObject
```

```
                    Dim oIInterface As IInterface =
objectIClassDefComp.GetInterface(interfaceDefIObject.UID, False)

                    If Not oIInterface Is Nothing Then
                        Dim oIProperty As IProperty
                        oIProperty =
oIInterface.GetProperty(propertyDefIObject.UID, False)
                        If Not oIProperty Is Nothing Then
                            If Not oIProperty.Value Is Nothing Then
                                ' Need to figure out what kind of
property it is.
                                Dim propertyDefIPropertyDef As
IPropertyDef = propertyDefIObject
                                Dim propertyTypeIObject As IObject =
propertyDefIPropertyDef.GetScopedByPropertyType()
                                Select Case
propertyTypeIObject.ClassDefIObj.Name
                                    Case "EnumListType"
                                        Dim propertyIEnumListProp As
IEnumListProp = oIProperty
                                        propertyIEnumListProp.ValueAsText
= True

toolObject.SetValue(mapPropertyIObject.Name, oIProperty.Value)
                                    Case Else

toolObject.SetValue(mapPropertyIObject.Name, oIProperty.Value)
                                End Select
                            Else
                                ' The property value is nothing. Our DAL
is not set up to deal with that so
                                ' set the property to blank

toolObject.SetValue(mapPropertyIObject.Name, "")
                            End If
                        End If
                    End If
                Else
                    ' This property is not mapped
                End If
            Next
        Else
            ' No properties are mapped on this class
        End If
    End Sub
```

19.7. Lab Exercise

Implement the following changes to the adapter…

- Change the SupportsFeature method in the adapter to respond true to the Mapping feature.

- Implement the GetMapSchemaFile method in the adapter to tell SmartPlant where your Tool Map Schema is.
- Change the RetrieveData function to use the GetMapClassForObject, CorrelateOrCreate, and RetrieveObjectProperties helper functions.
- Using the debugger, follow the flow of the RetrieveObjectProperties function to fully understand it's logic.

You may use the Chapter 19 – Lab 1 folder as a starting point. The completed lab exercise is available in the *Solution* project folder.

APPENDIX A

Schema Component Programming

A. IEFCommonUIApplication2

A. IEFCommonUIApplication2

If your application architecture prevents you from using the *IEFCommonUIApplication* interface, there is an additional interface named *IEFCommonUIApplication2* that will allow you to perform the same functions without the dialog boxes.

Property	Description
Connected As Boolean	Boolean indicating connection status to SmartPlant Foundation.
CurrentSPFProjectStatus As SPFProjectStatus	Current SmartPlant Foundation project status.
EFAdapter As IEFAdapter	Pointer to IEFAdapter.
LastErrorMessage As String	Error message generated by the last called method.
SPFPlant As String	SmartPlant Foundation plant.
SPFProjectName As String	SmartPlant Foundation project name.
SPFProjectUID As String	SmartPlant Foundation project unique ID.
SPFURL As String	SmartPlant Foundation URL.
ToolParameters As IEFToolParameters	Pointer to design tool parameters.

Method	Description
AutoRetrievalOptions() As Long	Displays the Automatic Retrieval Options dialog box.
ClaimObjects(oIContainer As IContainer, [sDocUID As String]) As Long	Claims objects after GetClaimContainer is called.
ClaimObjectsInXML(sDocUID As String, oClaims As DOMDocument) As Long	Claims objects using XML format.

Method	Description
ClaimObjectsInXML(sDocUID As String, oClaims As DOMDocument) As Long	Connects to SmartPlant Foundation/SMARTPLANT.
GetClaimContainer(oIContainer As IContainer, [sCompSchemaUID As String], [sDocTypeUID As String]) As Long	Creates schema component objects required for the claim operation.
GetComponentSchema(sCompSchemaUID As String, sCompSchemaFile As String) As Long	Returns a specified component schema file.
GetDocumentListContainerForAsBuiltPublish (sProjectUID As String, oDocsForAsBuiltPublishIContainer As IContainer, oSelectedDocIContainer As IContainer) As Long	Returns a container to identify as-built documents to publish to a project.
GetDocumentListContainerForPublish(oDocUIDs As IEFDocUIDs, oDocIContainer As IContainer, oNewDocIContainer As IContainer, oMetaIContainer As IContainer, oSelectedDocIContainer As IContainer, oSelectedMetaIContainer As IContainer) As Long	Creates the schema component objects required for publish.
GetInfoInXML(oInputArgs As DOMDocument, oOutput As DOMDocument) As Long	Fetches data in XML format for a given set of input arguments (generic method).
GetPublishableAsBuiltDocumentList(sProjectUID As String, oIContainer As IContainer) As Long	Returns a list of documents for which as-built data needs to be published to a project.
GetRetrievableDocumentList(oDocTypes As IEFDocTypes, oDocIContainer As IContainer, oMetaIContainer As IContainer, oSelectedDocIContainer As IContainer, oSelectedMetaIContainer As IContainer, [bOnlyDocsToBeRetrieved As Boolean], [bAllRevisions As Boolean]) As Long	Gets a list of retrievable documents.
GetRetrievableDocumentListByUID(sDocUID As String, oDocIContainer As IContainer, oMetaIContainer As IContainer, oSelectedDocIContainer As IContainer, oSelectedMetaIContainer As IContainer,	Gets a list of retrievable documents for a given document UID.

Method	Description
[bOnlyDocsToBeRetrieved As Boolean], [bAllRevisions As Boolean]) As Long	
GetSPFPlants(sSPFURL As String, oSPFPlants As ISPFPlants) As Long	Returns the names of plants in a SmartPlant Foundation URL.
GetTitleBlockInfo(sDocUID As String, sDocRev As String, oTitleBlockInfo As IContainer) As Long	Returns title block information for a document revision.
GetUnclaimContainer(oIContainer As IContainer, [sCompSchemaUID As String], [sDocTypeUID As String]) As Long	Creates the schema component objects required for unclaim operation.
Publish(oSelectedDocIContainer As IContainer, oSelectedMetaIContainer As IContainer, sLogFile As String) As Long	Processes publish after GetDocumentListContainerForPublish is called.
PublishAllAsBuiltDocuments(sProjectUID As String, sLogFile As String) As Long	Publishes as-built documents to a project.
PublishSpecifiedAsBuiltDocuments(sProjectUID As String, oDocIContainer As IContainer, sLogFile As String) As Long	Publishes the specified as-built documents to a project.
Register(sToolID As String, sToolDescription As String, sToolEFAdapterProgID As String, oToolParameters As IEFToolParameters, sSPFURL As String, sSPFPlant As String, sSignature As String, [oPBSIContainer As IContainer]) As Long	Registers a tool with SmartPlant Foundation.
ReleaseClaimContainer() As Long	Rolls back the claim operation initiated by GetClaimContainer method.
ReleasePublishDocumentListContainer() As Long	Releases containers created in the GetDocumentListContainerForPublish method.
ReleaseRetrieveContainer() As Long	Releases containers created for the retrieve operation.

Method	Description
ReleaseUnclaimContainer() As Long	Rolls back the unclaim operation initiated by the GetUnclaimContainer method.
Retrieve(oSelectedDocIContainer As IContainer, oSelectedMetaIContainer As IContainer, sLogFile As String) As Long	Processes the retrieval of selected documents.
UnclaimObjects(oIContainer As IContainer) As Long	Unclaims objects after GetUnclaimContainer is called.

Many of the properties and methods are the same as those for the IEFCommonUIApplication interface; however, there are no methods to display the Common UI dialog boxes, and there are additional methods so that you can provide the functionality of those dialog boxes from within your application. We will not be using this interface during this course.