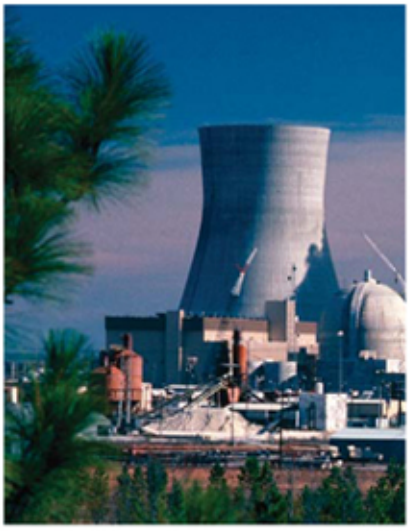
SmartPlant P&ID 2007

Automation Programming with VB Course Guide

Process, Power & Marine


INTERGRAPH

Version 2007 April 2007 DPID2-TP-100024A

Copyright

Restricted Rights Legend

Intergraph Corporation
Huntsville, Alabama 35894-0001

Warranties and Liabilities

Trademarks

Table of Contents

Preface

This document is a user's guide for SmartPlant P&ID 2008 automation programming with VB course.

Send documentation comments or suggestions to PPMdoc@intergraph.com.

CHAPTER 1: REVIEW OF VISUAL BASIC CONCEPTS

1. OBJECTIVES

In this chapter you will review:

◊ How to write simple standard executables

◊ The Visual Basic environment

◊ Some object-oriented concepts

2. ADVANTAGES OF USING VISUAL BASIC

◊ Visual Basic is an extension of the BASIC programming language utilizing the object-oriented technology.

◊ It provides an easy-to-use development environment that simplifies the creation of applications requiring graphical user interfaces.

◊ It supports OLE Automation programming because it is COM compliant.

  ◊ COM (Component Object Model) specifies binary level interfaces how components are created and how client applications connect to components.

  ◊ OLE (Object Linking and Embedding) Automation further specifies interfaces that manipulate an application's objects from outside of the application.

3. CREATING A STANDARD EXECUTABLE BASIC PROGRAM USING VB

3.1. *Module-Level Code*

◊ Standard Basic code is written in Modules. A module is a code segment that defines functions and subroutines.

3.2. *Programming features in VB*

3.2.1. Start-up Object

◊ Start-up function is the Sub Main() in the Module

3.2.2.Loop structures

◊ Do …Loop While

- ◊ Do…Loop Until
- ◊ Do While …Loop
- ◊ Do Until …Loop
- ◊ For … Next
- ◊ Exit statement can be used to exit a Do or For loop

3.2.3. Conditional IF structure

- ◊ IF … THEN
- ◊ IF … THEN … ELSE … ENDIF
- ◊ IF … THEN … ELSEIF … THEN … ELSE … ENDIF

3.2.4.Select Case Statements

- ◊ SELECT CASE .. CASE .. CASE … END SELECT

3.2.5.String concatenation

- ◊ Use the '&' symbol to concatenate strings into a variable.

3.2.6.Help feature

- ◊ Place the cursor on a keyword and press the F1 key to obtain help on the keyword

3.3. *Debugging a standard executable*

- ◊ Use the Breakpoints to stop a program.
- ◊ Use Watch expressions
- ◊ Step into, Step over, and Step out of.
- ◊ View the Call Stack.
- ◊ Run commands in the Immediate window.
- ◊ Examine objects in the Locals window.
- ◊ The Debug object with the Print method.
- ◊ The Err object and its properties: Number, Description, Source, Clear, Raise.
- ◊ The On Error statement.

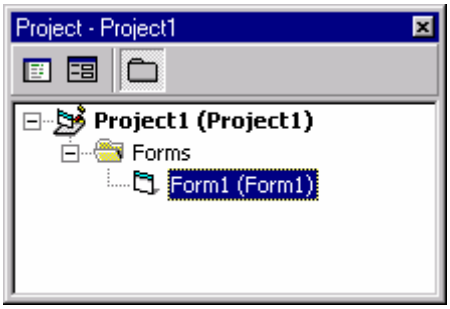
3.4. *Compile and Run a Standard executable*

- ◊ Select File->Make->…
- ◊ Enter the name of the executable file

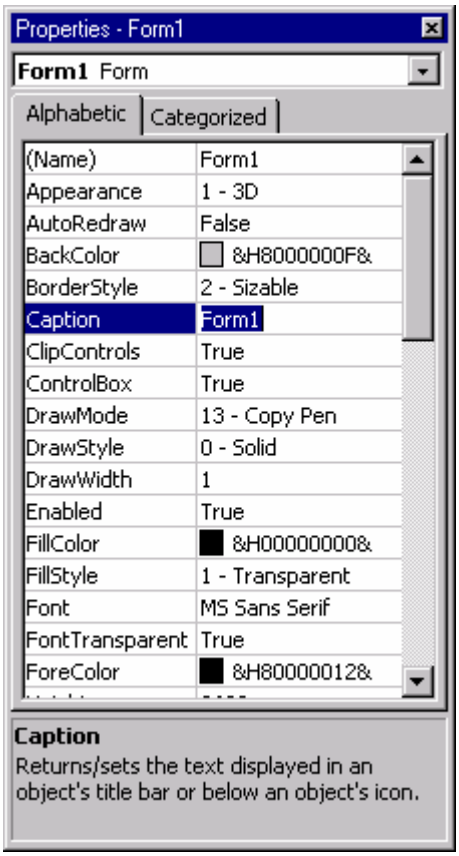
4. VISUAL BASIC ENVIRONMENT

4.1. *Toolbox*

4.2. *Project Explorer Window*



4.3. *Properties Window*



4.4. *Object Browser for Browsing the Libraries*

The Object Browser can be opened from the Standard Toolbar or from the View command on the menu. The Object Browser is used to examine the methods and properties of the classes in object libraries.

4.5. *Immediate, Locals, and Watch Window for Debugging*

The Immediate, Locals, and Watch Windows can be opened from the Debug Toolbar or from the View command on the menu. These windows are used to examine the values of variables and objects at run time.

4.6. *References*

The Project/References… command pulls up a dialog to select libraries that can be referenced into the current project. By referencing these libraries, the type information of the classes in these libraries are available to the programmer during code development and compilation.

4.7. *Components*

The Project/Components… command pulls up a dialog to select the ActiveX components that can be included into the current project. ActiveX components can be used to provide user-interface and code level functionalities to the project.

4.8. *Project Properties*

◊ Convert between standard EXE and Active-X dlls.

◊ Select start-up component.

◊ Set compatibility.

5. USING VISUAL BASIC

5.1. *Controls*

◊ Controls are predefined classes with graphical representations.

◊ Controls can be added to the project by dragging onto a form from the Toolbox.

◊ Every instance of a control has a unique name within the form

5.2. *Forms*

◊ Forms are like Controls, with properties, methods, and events.

◊ Forms are containers for Active-X controls.

5.3. *Procedures*

◊ Two types: Event Procedures and General Procedures

◊ General Procedures include Function procedures and Subroutine procedures

5.4. *Variables and Constants: Declaration, Scope, and Life*

◊ Always use the keywords Option Explicit at the top of every module. This will ensure that variables are always defined before they are used.

◊ Use Dim, Private, or Public to declare variables.

◊ Constants are declared with the CONST keyword

- ◊ Procedure-Level Variables are declared in a procedure and is recognized only within the procedure. Procedure-level are created at the start of the procedure and destroyed at the close of the procedure.
- ◊ Form-Level Variables are declared outside of procedures but inside a General Declarations section of the form and is global within the form. Public variables alone will be accessible to procedures outside of the form as a property of the form. They have to be referenced using the name of the form, such as frmMyForm.Property1. They are available from the instant the form is created until it is destroyed (`set frmMyForm = Nothing`).
- ◊ Standard Module-level Variables are declared within the General Declarations section of the Module. Private variables are available only within the Module, while Public variables are available as global variables within the whole project.

5.5. *DataTypes*

- ◊ Data Types include Integers, Longs, Double, String, Variant, etc.
- ◊ Variants hold all datatypes but not objects. A variant can have a NULL value, indicating that it is undefined.

6. FILES IN THE VISUAL BASIC PROJECT

6.1. *Project*

- ◊ A Project consists of all of the form modules, standard modules, class modules, controls, references, and settings required to compile and run the application.
- ◊ It is saved as a *.vbp file.
- ◊ A project is synonymous with Library or Program, depending on the context.

6.2. *Standard Module*

- ◊ It is a standard code module.
- ◊ It is saved as a *.bas file.

6.3. *Class Module*

- ◊ It is a template for an object.
- ◊ An interface is an abstract class.
- ◊ It is saved as a *.cls file.

6.4. *Form Module*

- ◊ A Form is a Class definition.
- ◊ It is a container for controls.
- ◊ It has properties and methods like other classes.
- ◊ It has pre-defined Events just as for Controls.
- ◊ It is saved as a *.frm file.

7. OBJECT-ORIENTED CONCEPTS

7.1. *Object*

An object is a collection of data with related functions and subroutines that can be grouped as a single unit.

7.2. *Class*

- ◊ A class is a template for an object. It contains the structure and definitions of all of the parts of an object.
- ◊ A class is initialized when an object instance of the class is created. Eg:
  `set obj1 = New Class1`
- ◊ A class is terminated when the object instance of the class is set to nothing. Eg:
  `set obj1 = Nothing`
- ◊ The name of the class can be edited in the Property Box for the class module. This name can be different from the filename under which the class module is saved.

7.3. *Methods and Properties of Classes*

- ◊ Methods and Properties of a class are those variables and procedures that cannot be accessed other than within the context of an object instance of the class.
- ◊ The methods and properties of a class are accessed using the dot (.) extension to the class name. For instance, if the class `Object1` has a property called `Name` and `obj1` is an object instance of that class, then the property can be accessed using the syntax `obj1.Name`.
- ◊ Properties and Methods can be either Public or Private, depending on whether they must be visible to objects outside this class.

- ◊ Properties can also be defined as procedures to provide more access and validation control. Such procedures are prefixed by the keyword 'Property'.
- ◊ Properties have three procedure definitions: Get, Let, and Set.
- ◊ The Get procedure returns the value of the property and is called when the code is reading the value of the property such as in `Debug.Print obj1.Name`
- ◊ The Let procedure is called by VB when the property value is being assigned a value in the code. Eg. `Obj1.Name = "Object 1"`
- ◊ A Get procedure without the implementation of the Let procedure creates a Read-only property.
- ◊ A Set procedure is just like the Let procedure, except that it is used associate an object variable to an object. In this case the Set keyword must be used. Eg. `set obj1 = New Object1`
- ◊ Sample Code:

```
Public Property Let Name(vdata As String)
    strname = vdata
End Property

Public Property Get Name() As String
    Name = strname
End Property

Public Property Set Myobject(vdata as object)
    Set mvarMyObject = vdata
End Property
```

- ◊ Methods are prefixed by either the keyword 'Function' or 'Sub'. Subroutines do not have a return value.
- ◊ Events of a control are suffixed into the name of the control with an underbar (_). Eg. Command1_Click() is the Event method for the Click event of the Command1 command button.
- ◊ Every class has a Class_Initialize() and a Class_Terminate() event that is executed at the initialization and termination stages of the class, respectively.

8. CREATING CLASSES

8.1. *Create a Class Module*

- ◊ Class modules contain code that define the properties and methods of the class.
- ◊ Classes can be created using the Class Builder Utility under the Add-in Manager.

8.2. *Create Properties and Methods*

◊ Properties and Methods can be added using the Class Builder Utility or they can be manually edited into the class module.

9. LAB

Optional Lab 1: Write a simple VB code and debug it.

Optional Lab 2: Write a simple VB code using a Form.

10. REVIEW

CHAPTER 2: CREATING AND RUNNING ACTIVE-X CLIENT COMPONENTS

1. OBJECTIVES

In this chapter you will review the steps needed to develop:

◊ Active-X clients that use Automation objects

2. INTRODUCTION

◊ A client application is code that accesses the properties and methods of other Automation objects.

◊ The Object Variable must be declared using the ProgID of the object. The ProgID is made up of the ProjectName and the ClassName concatenated with the dot (.) symbol. Eg. “Excel.Application”

◊ Standard applications, such as MS Word and MS Excel, can provide Automation objects that can be accessed by client applications.

◊ Client applications get information about Automation objects through the type libraries associated with the Automation object. Type libraries provide information about the Interfaces supported by the object, descriptions of the properties, methods, and events provided by the object, the return types and parameter types of the methods and events, etc. This information for Active-X components can be viewed through the Object Browser.

◊ The structure of the inter-relationships of objects within the type library is depicted by the Object Models.

3. CREATING A CLIENT COMPONENT

◊ Create a Standard executable VB Project

◊ Set a reference to the component library

◊ Declare and create an object variable representing the component

◊ Use the object’s methods and properties

4. DECLARING AND CREATING AN OBJECT VARIABLE OF SERVER COMPONENT

◊ Use CreateObject(ProgID) method to instantiate an object.

```
Dim obj as Excel.Application
Set obj = CreateObject("Excel.Applicaton")
```

◊ Use the New keyword.

```
Dim obj as Excel.Application
Set obj = New Excel.Applicaton
```

5. LAB

Optional Lab 3: Using object browser to view Automation objects.

Optional Lab 4: Write VB client application to access Microsoft Excel's Automation objects.

6. REVIEW

CHAPTER 3: CREATING AND RUNNING ACTIVE-X SERVER COMPONENTS

1. OBJECTIVES

In this chapter you will review the steps needed to develop:

◊ Active-X server components in the Visual Basic (6.0) development environment.

2. ACTIVE-X SERVER COMPONENTS

◊ Active-X server components are libraries of objects that can be used by client components.

◊ These have a Project Type of Active-X DLL.

3. CREATING AN ACTIVE-X SERVER COMPONENT

◊ Create an Active-X dll project

◊ Create a Class Module

◊ Create functions and properties (defined in the abstract class) in the new class.

◊ Compile the component or run the VB project to register the class for debugging.

4. CREATE A CLIENT APPLICATION THAT USES THE ACTIVE-X SERVER

◊ Create a standard executable which references the Active-X Server

◊ Dimension and create an instance of the class.

◊ Access functions and properties of the class.

5. LAB

Optional Lab 5: Create an active-x server and a client application.

6. REVIEW

CHAPTER 4: CREATING ACTIVE-X SERVER COMPONENTS SUPPORTS INTERFACES

1. OBJECTIVES

In this chapter you will review the steps needed to develop:

◊ Active-X server components with an interface in the Visual Basic (6.0) development environment.

2. ACTIVE-X SERVER COMPONENTS

◊ Active-X server components are libraries of objects that can be used by client components.

◊ These have a Project Type of Active-X DLL.

3. INTERFACES

◊ Interfaces are abstract classes. Abstract classes act as templates because they provide function and property definitions without any implementations.

◊ An interface specifies the functionalities guaranteed by an object.

◊ Interfaces allow objects to grow in functionality without breaking the earlier functionalities, as long as they continue to implement the original interfaces.

4. CREATING AN ACTIVE-X SERVER COMPONENT WITH INTERFACE

4.1. *Create an abstract class for the interface*

◊ Create an Active-X dll project.

◊ Create a class module with the name of the abstract class and interface.

◊ Create the functionalities of the interface by adding methods and properties to the abstract class. No implementation code must be included.

◊ Compile the code or run the VB project to register the class for debugging .

4.2. *Create an implementation of the abstract class or interface*

◊ Create an Active-X dll project that references the abstract class created above.

- ◊ Use the 'Implements' keyword followed by the name of the abstract class inside the implementation class module. This class then implements the interface described by the abstract class.
- ◊ Implement the functions and properties (defined in the abstract class) in the new class.
- ◊ Compile the component or run the VB project to register the class for debugging.

4.3. *Create a client application that uses the implementation*

- ◊ Create a standard executable which references the abstract class and the implementation class.
- ◊ Dimension variables of the abstract class as well as the implementation class.
- ◊ Set the abstract class variables to point to the existing implementation class objects.
- ◊ The functionalities available at each object depends on its class definition, even though they point to the same object.

5. LAB

Optional Lab 6: Create an interface, an implementation, and a client application.

6. REVIEW

CHAPTER 5: DATA MODEL - FUNDAMENTALS

1. OBJECTIVE

In this chapter, you will learn

◊ the Data Model terminology

◊ how to interpret the UML diagrams

2. DATA MODEL IN OBJECT-ORIENTED TERMINOLOGY

◊ Every software works with data using methods or routines. Data and methods can be grouped to form self-contained objects that interact with other objects.

◊ A class is a template for an object. Multiple objects of a given class can be used in a project. A project may also include a number of different classes. These classes can be related to one another. The relationship among the classes represents the Data Model.

◊ The Static Data Model describes the nature and inter-relationships of the various classes in a project. The Data Model can be represented as a Class Diagram using Unified Modeling Language (UML). UML is an emerging language for creating models of object-oriented computer software.

3. DATA MODEL REPRESENTED USING UML

3.1. *Object Classes*

◊ The Data Model deals with classes of objects.

◊ An object class is a template for an object while an instance refers to a particular object created during the use of the software.

◊ An object is made up of attributes (member variables) and operations (member functions and subroutines).

◊ An object class is represented by a box called the class icon. It is a rectangle divided into three compartments.

◊ The topmost compartment contains the name of the class. Typically, the second compartment lists the attributes of the class, and the third compartment lists the operations. Normally, only relevant attributes and operations are listed in these compartments for space considerations.

3.2. *Relationships*

- ◊ Object classes in the Data Model are normally related to other object classes.
- ◊ These relationships include simple associations, aggregate relationships, and inheritance and are always denoted by a relationship line.
- ◊ Most of the relationships are maintained by member variables. One member variable in one of the objects will represent the unique identifier of the related object.

3.2.1. Association (Open Arrow)

- ◊ A simple association is a reference to an object inside another object.
- ◊ The direction of the open arrow indicates the direction of the navigation. The object class that maintains the information about the relationship is at the base of the arrow.
- ◊ Instances of both objects must exist before the relationship can be created.

3.2.2. Inheritance (Closed Triangular Arrow)

- ◊ In the inheritance relationship, the arrow is at the parent object, which is the superclass.
- ◊ The child object, the sub-class, derives from the parent class and inherits all of the attributes and operations of the parent class.
- ◊ The sub-class is a more specific kind of the superclass. The derived classes retain pointers to their parent classes.

3.2.3. Aggregation and Composition (Diamonds)

- ◊ Diamonds indicate Parent-Child relationships. The entity with the diamond is the Parent.
- ◊ The child exists only if the parent exists. For example, a Nozzle must have a parent Equipment. If the parent Equipment is deleted, the child Nozzle cannot exist.

3.2.4. Text

- ◊ The text on the relationship line shows the nature of the relationship.

3.2.5. Numbers

- ◊ Numbers at the ends of the relationship line indicate “multiplicity” or the number of instances of the respective object class that can exist in the relationship. For instance, the numbers between ModelItem and Representation indicate that one ModelItem may be related to (associated with) multiple (0…*) Representations. Some ModelItems do not have any representations (0), while some can have multiple representations (*).

4. REVIEW

CHAPTER 6: LOGICAL MODEL AUTOMATION (LLAMA)

1. OBJECTIVES

In this chapter, you will learn

◊ interpret the Logical Model Automation classes and properties from the Data Model

2. INTRODUCTION

The Logical Model Automation (Llama) is primarily an object model based on the SmartPlant P&ID DataModel. In order to facilitate proper use of Automation, Llama also incorporates some non-DataModel items.

3. ITEMS FROM THE DATAMODEL

3.1. *Object Classes*

◊ Each object class in the DataModel has two Visual Basic (VB) classes. The first represents the object class while the second represents a collection of the object class. The names of these classes are prefixed by an "LM". The LM stands for the Logical Model. For example, the PipeRun object class in the DataModel has two classes, namely LMPipeRun and LMPipeRuns, in VB.

◊ Llama does not provide classes for the *Join objects. These objects contain connection informaton for a many-to-many relationship between two object classes. For instance, the PlantItemGroupJoin object class holds connection information between the PlantItem and PlantItemGroup object classes, which are related through a many-to-many relationship. Llama provides this information in the same way that it supports relationships. This will be discussed in the section on Relationships.

◊ Certain relationships shown in the Data Model are not implemented in SPPID. For instance, the relationship between Nozzle and PipeRun and the many-to-many relationships indicated by the PipeRunJoin and SignalRunJoin tables are currently not implemented.

3.2. *Attributes*

◊ Every object has one or more attributes.

◊ Although many objects may use similar attributes (e.g. SP_ID), the attributes of an object are unique.

◊ Attributes from the DataModel are implemented in three different ways in Llama, depending on the datatypes of the attribute. These are the simple datatypes, the enumerated datatypes, and the unitted datatypes.

3.2.1. Simple Data Types

◊ The simple datatype attributes, such as string, date, long, etc, are implemented as standard Properties of the VB class.

◊ The Let and Get functions are typically defined for the Property, enabling the VB user to both assign the value and to read the value.

3.2.2.Enumerated Data Types

◊ The enumerated attributes are codelisted attributes. Each codelist has a number (an integer part) and a name (a string part) and these are listed in the 'Enumerations' table in the DataDictionary. The members of each codelist are listed in the 'codelists' table in the DataDictionary. They have a set of predefined values with corresponding indices for each valid value. The values and the indices together form the codelist associated with the attribute.

◊ In Llama, the enumerated attributes are named after the codelists that they represent.

◊ Each enumerated attribute has two Properties associated with it in Llama. One has the name of the attribute, and it can be used to either get or set the name (string part) of the value. The other has the name of the attribute suffixed with the word "Index", and can be used to get or set the integer part (index) of the value. The index zero (0) indicates a null value.

3.2.3.Unitted Attributes

◊ The unitted attributes are the formatted attributes.

◊ They are composed of a numeric value and a text string depicting the unit of measure. The numeric value is an arbitrary number selected by the user, but the unit of measure must be recognized by SmartPlant P&ID as one of the defined formats.

◊ In Llama, the unitted attribute has two Properties associated with it. One has the name of the attribute, and it is the concatenation of the floating point value and the unit of measure as a single string. It can be used to set and get the value of the attribute in string form. The other property has the name of the attribute suffixed with the word "SI" and it yields the numeric value converted into the SI units for that quantity. This latter function is read-only and has only the Get part of the property.

3.3. *ItemAttribution*

◊ The relationship between objects allow objects to own attributes that are unique to other objects through various relationships. These derived attributes are part of the 'Item Attributions' of that item.

◊ The inheritance relationship naturally includes the attributes of the superclass into the Item Attributions of the subclass. For instance, the ItemAttributions of the Equipment class include attributes of PlantItem because of the inheritance relationship between Equipment and PlantItem.

◊ An association relationship between two items can permit one or more attributes of one class to be Item Attributes of the other. These Item Attributes have to be explicitly added in order for the attribute to be available from the object. The relationship between Case and ModelItem allows Equipment to have some Case attributes in its ItemAttributions list because of the ItemAttributions created between the two classes. However, other object classes that inherit from ModelItem do not have to own the same Case attributes and therefore do not have these attributes as part of their itemAttributions list.

◊ Each item attribution has a Name, a DisplayName, and an ID among other properties. If the item attribution is displayable through the property grid, then the DisplayName will be displayed in the property grid for the particular item.

◊ The ItemAttribution name is generally based on the path used to navigate to the property from a given item. However, this format is not a requirement. The only requirement is that the name must be unique for a given object class.

◊ Item Attributions not directly owned by the item are maintained in memory for performance reasons. Starting V3.0, first time user tries to access these Item Attributions will make these Item Attributions available for the object, no special trigger is needed. In addition, if user get the object using PIDDatasource, then all Item Attributions will be in the collection by default.

3.4.

3.5. *Relationships*

◊ Llama implements the relationships defined in the DataModel through properties. These properties appear in the 'LM' class of the object class. The type of property that appears depends on the multiplicity of the specific end of the relationship.

3.5.1. One-to-Many Relationship

◊ In a one-to-many relationship, the LM class on the 'one' end of the relationship will have a property that returns a collection of the item that appears on the 'many' end of the

relationship. As an example, in the one-to-many relationship between Equipment and Nozzles, LMEquipment has a property called Nozzles, which returns all of the nozzles associated with the given instance of the LMEquipment.

◊ In the item on the many end of the relationship, there is a property that returns the unique item that it is related to. In the above example, the LMNozzle has a property that returns the EquipmentObject that it is related to.

3.5.2.Inheritance Relationship

◊ Inheritance relationships are visible through the 'additional' properties inherited by the sub-class LM object:

◊ The sub-class LM object contains every property of the superclass. For example, LMEquipment has a property called ItemTag, which it inherited from its superclass, LMPlantItem. LMVessel, which is a sub-class of LMEquipment, also inherits the ItemTag property from LMPlantItem.

◊ The subclass LM object inherits relationships of its superclass. For example, LMPlantItem has a property called Cases, which returns a collection of Cases, because it inherits the one-to-many relationship between its superclass ModelItem and Case in the DataModel.

3.5.3.Many-to-Many Relationships

◊ In a many-to-many relationship, the LM classes associated with both the object classes have properties that return a collection of the other LM class. For example, LMPlantItem has a property called PlantItemGroups, and LMPlantItemGroup has a property called PlantItems.

3.5.4.Group Relationships

◊ In a group or parent-child aggregation relationship, a child object has only one parent while a parent can have many children. Consequently, most parent-child aggregations are typically one-to-many relationships.

◊ Llama provides a property that returns a collection of the 'many' object in the group relationship. Although some parent-child relationships have a one-to-one relationship or even a one-to-none relationship, a collection is still provided. For example, the PipingComp and InlineComp have a parent-child aggregation relationship with multiplicity of 0..1. This implies that an InlineComp can have at most one PipingComp parent and the PipingComp can have at most one InlineComp child. Although the latter condition suggests a single InlineComp for a PipingComp, Llama still supports a property that returns a collection of LMInlineComps for a given PipeRun. This collection may contain only one object, consistent with the multiplicity of the relationship.

4. ITEMS OUTSIDE OF THE DATA MODEL (LMA)

In addition to the object classes provided in the DataModel, Llama provides some additional classes to enable the use of Automation.

4.1. *LMADatasource*

- ◊ The LMADatasource is the primary object in Llama because it provides the connection to the database.
- ◊ The LMADatasource establishes connection to the database using the Plant Name in SPManager. The ProjectNumber is the property that is set equal to the active SPManager Plant Name as default. To switch to a new SmartPlant P&ID Plant, user must set a the exactly name of the Plant to the ProjectNumber of the LMADatasource
- ◊ There are three sources to get LMADatasource. First is by setting a new LMADatasource, second is get PIDDatasource from the active drawing, third is get from SPPID when running validation program.
- ◊ A New LMADatasource must be created in a standalone project, unless the datasource can be obtained from another process. In Calculation/Validation, a New Datasource does not have to be created because the Active-X dll receives an initialized LMADatasource from the SPPID.
- ◊ The three properties IPile, Pile, and PIDMgr have non-Llama return types and can be used to get the connections to the Data Coordinator and PIDObjectManager or to set the connections to existing connections. The typical Llama user will not have to use any of these properties. These are for advanced uses.

4.2. *Labs*

Lab 1: Initialize LMADataSource and access its properties.

Lab 2: Change Site and Plant

Lab 3: Access ItemTypes.

4.3. *LMAItem*

- ◊ The LMAItem is a generic item in Llama. It has an interface that is supported by every object class in Llama. Therefore, an LMVessel can be converted to LMAItem.
- ◊ Every LMAItem represents the object class it was converted from. Typically, one would use the ItemType to determine the type of item to 'get' from the datasource. For instance, LMVessel as an LMAItem will yield an ItemType of 'Vessel' while the same vessel item when retrieved as an LMEquipment and then converted into LMAItem will yield an ItemType of 'Equipment'. However,

this is not TRUE anymore since V4. In V4, all model item object will be obtained in its concrete level. For example, even retrieved as n LMEquipment, its ItemType is “Vessel”. In addition, the object will have all ItemAttributions collection as its concrete object.

4.4. *LMAAttribute*

◊ An LMAAttribute is a generic attribute in Llama and uses the ItemAttribution. It has a Name and a Value. These are applicable for all types of attributes.

◊ Additional properties such as Index and SIValue are applicable for enumerated (codelisted) and unitted (formatted) attributes, respectively.

◊ The ISPAttribute is a property is reserved for advanced uses and is not supported in Llama.

◊ Typically, an LMAAttribute can be identified from the LMAAttributes collection by using the Name as listed in the Item Attributions table.

4.5. *Labs*

Lab 4: Identify an Item and read its properties.

Lab 5: Modify the properties of an Item.

Lab 6: Init Objects Read Only.

Lab 7: Rollback.

Lab 8: Propagation.

Lab 9: Get LMAItem from LMVessel.

Lab 10: Access LMAAttributes Collection.

Lab 11: Access ItemAttributions in Details.

Optional Lab 7: Modify Property Based on Construction Status

4.6. *LMACriterion*

The LMACriterion is an object class that is typically used with the LMAFilter class.

◊ A criterion relates an attribute with a value. The criterion can be fully defined by setting values for at least three properties, namely SourceAttributeName, ValueAttribute, and Operator.

◊ The SourceAttributeName is the name of the attribute from the ItemAttributions table.

- ◊ The ValueAttribute is the value of the attribute, it also can be a string containing a symbol such as "%","_" etc.
- ◊ The Operator indicates the relationship between the attribute and its value. The operator is a string containing a symbol such as "=", "<", ">=", "like", "is" etc.
- ◊ The SourceAttributeID is the ID of the ItemAttribution.
- ◊ The SIValue can be specified in the case of formatted attributes.
- ◊ The ValueDisplayString is value of formatted attributes as displayed in the PropertyGrid.
- ◊ The Conjunctive must be set to True if this LMACriterion must be AND-ed with other LMACriterions.

4.7. *LMAFilter*

- ◊ LMAFilter.Criteria.Add Criterion adds the Criterion into the Criteria collection in the Filter.
- ◊ LMAFilter.Criteria.AddNew without an argument will create an new LMACriterion to the filter.
- ◊ The LMAFilter.Criteria collection has default indices starting with 1.
- ◊ If a specific *vntindexkey* is to be assigned, then it can be given as LMAFilter.Criteria.AddNew("3") or LMAFilter.Criteria.AddNew("Special").
- ◊ A counter key is still available for an LMACriterion based on the number of Criteria that existed prior to its creation.
- ◊ If a Criterion is deleted from the list, the counter key is renumbered but the list maintains the same sequence.

4.8. *LMAEnumAttList*

- ◊ An LMAEnumAttList in Llama is referred to a SelectList data.
- ◊ Property EnumeratedAttributes is a collection of SelectList value of this SelectList Data.
- ◊ User can get a collection of LMAEnumAttList from LMADataSource.CodeLists property.

4.9. *LMAEnumeratedAttribute*

- ◊ An LMAEnumberatedAttribute in Llama is referred to a SelectList Value in a Select data.
- ◊ Properties of LMAEnumberatedAttribute, such as Name and Index will return the Name and Index of a select value. This will be very useful when Llama user creates a LMAFilter in program, since Llama only takes Index value in Criterion.

5. GENERAL ISSUES

5.1. *Properties and Methods of Collections*

- ◊ *Property Count*: returns the number of items in the collection. For an empty collection, the Count is zero.
- ◊ *Property Item(ID as long)*: returns an item of the type that this collection holds. The argument is the IndexKey of the item. The IndexKey is generally the SP_ID of the item. If the IndexKey is wrong, the returned item is a Nothing.
- ◊ *Property Nth(index as long)*: selects the item in the sequence that it is stored in the collection.
- ◊ *Property Datasource*: returns the Datasource that was used to create the collection. This is important when a number of datasources are in use, each pointing to different plants or databases.
- ◊ *Property TypeName As String:* Returns the item type name of the items in the collection.
- ◊ *Sub Remove(ID as long)*: removes the item with the given ID, if it exists in the collection.
- ◊ *Sub Collect([DataSource As LMADataSource], [Parent As LMAItem], [RelationshipName As String], [Filter As LMAFilter]):* collects items of the kind represented by the collection from the specified DataSource. The filter can be used to identify specific properties for the collected items. The RelationshipName is the name given on the relationship lines in the DataModel.
- ◊ *Sub Clear():* Clears the collection.
- ◊ *Sub AddCollection(Items As …)*: Add all items from the indicated collection into the current collection.
- ◊ *Function Add([Item As …]) As …:* Adds an item to the collection. Note that the argument and the return value are of the added Llama object.
- ◊ *Function AsLMAItems() As LMAItems*: Returns the LMAItems interface of the collection.
- ◊ The LMACriterions collection has a special *Function Add(NewObject As LMACriterion)* which returns a Boolean.

6. LABS

Lab 12: Access an item using filters with single criterion.

Lab 13: Access items using filters with multiple criteria.

Lab 14: Access items using filters with criterion on Select List Data.

Lab 15: Using Compound Filter

Lab 16: Access all filters defined in the plant.

Lab 17: Access SelectList Data.

Optional Lab 8: Serach Items and Modify Property

7. REVIEW

CHAPTER 7: SPPID DATA MODEL

1. OBJECTIVES

In this chapter, you will learn

◊ the general structure of the SmartPlant P&ID Data Model and its implementation in the Llama object model

◊ highlights the relationships between the classes that need additional attention

2. INTRODUCTION

The objects exposed through Llama span all the three schemas used in the database. Objects in each grouping may come from different schemas. However, the object model relationships allow the user to access the related objects and obtain the appropriate properties without being concerned about the schema they belong to. Therefore, the most important task for the user of Automation is fully understand the various relationships between objects. This chapter addresses some objects and relationships that requires special attention from the user.

3. MODEL DATA

The Model Data maintains the engineering data. It is centered around the ModelItem and the PlantItem, which is also a ModelItem. Refer to the Data Model diagram for the relationships between the various objects. This section describes only the specially noteworthy and the not-so-obvious characteristics of the objects.

3.1. *ModelItem*

The ModelItem can own one or more instances of Alias, History, Event, Note, Source, and Status. In addition, it can own one or more Representations from the Drawing Data Model. Currently the software creates History records of HistoryType 'Creation' and 'Last Modification'. The user should not modify any of the History records maintained by the system nor create new ones of these types. However, the user can create new History records of a different type after creating new types in the datadictionary, if necessary. The Automation user can also create new instances of Alias, Event, Note, Source, and Status.

In addition, a ModelItem can own one or more Cases. These Cases are maintained by the software using the information provided in the DataDictionary and the ItemAttributions. The Automation user should not attempt to add new Cases, CaseProcesses, or CaseControls through code.

The ModelItem is the superclass from which all other Model Data classes are derived. The immediate subclasses of the ModelItem are the ItemNote, PipingPoint, PlantItem, SignalPoint, and OPC. Of these, the most significant is the PlantItem.

3.2. *Labs*

Lab 18: Read History Property of ModelItem.

Lab 19: Read Status Property of ModelItem.

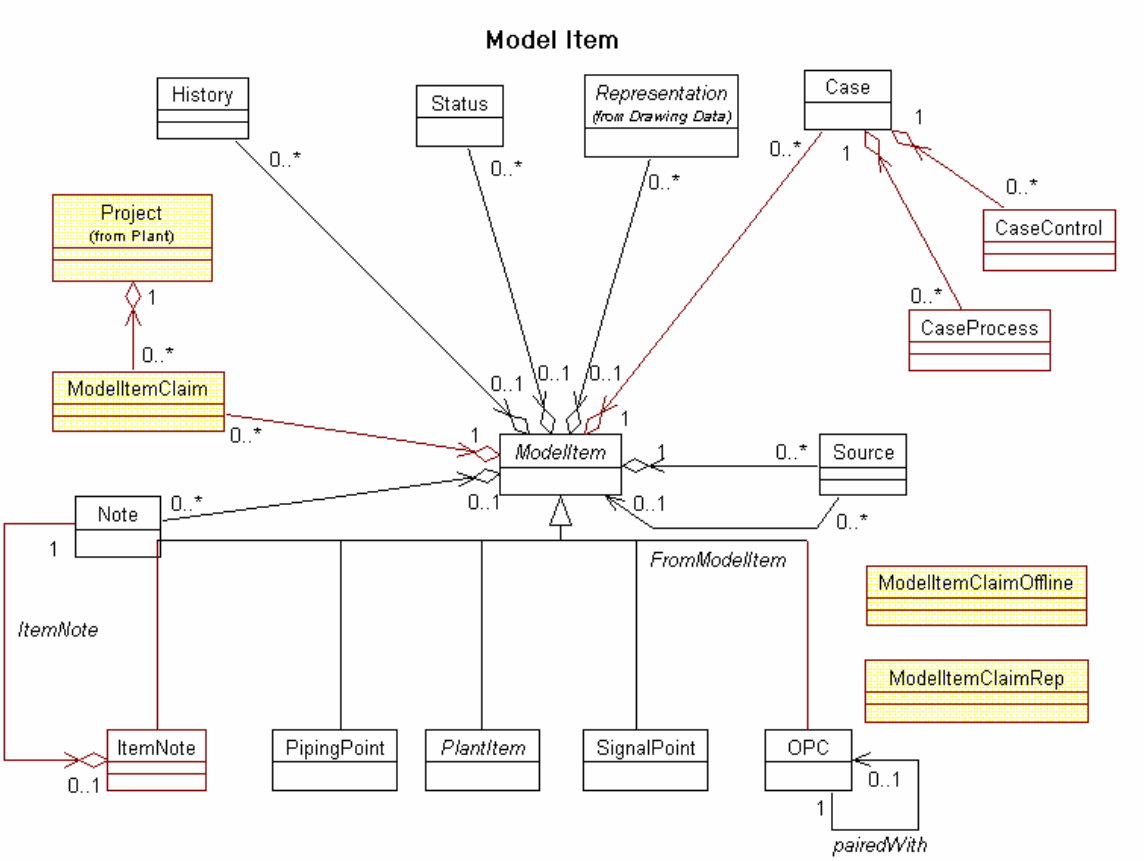
Lab 20: Read Case Property of ModelItem.

Lab 21: Access ItemNote.

Lab 22: Access OPC.

Lab 23: Filter for Histories

Optional Lab 9: Modify Case Process Data



Model Data: Model Item

3.3. *PlantItem*

Most items that are placed on drawings can be traced back to PlantItem. Subclasses of the PlantItem are Equipment, Nozzle, PipingComp, Instrument, Pipeline, Piperun, SignalLine, SignalRun, and PlantItemGroup.

3.3.1. PartOfPlantItemID and PartOfPlantItemObject

Returns the PlantItem that this current PlantItem is a Part Of. Examples are implied items, TEMA ends, and trays. An implied item is related to its parent item through this relationship. It also has a PartOf type = 'Implied'. A TEMA end is a part of the TEMA Shell that it is placed on and has a PartOf type = 'Composite'. Although a tray is a legitimate part of the vessel it is placed on, it does not have a PartOf type value. The value is NULL for a tray.

3.3.2. PlantItemGroup and PlantItem

PlantItemGroup is a subclass of PlanItem, concrete PlantItemGroup includes: Package, SafetyClass, System, InstrLoop, PlantItemGroupOther, and AreaBreak. PlantItemGroup has many to many relationship with PlantItem. For example, a Package can have two vessels and many piperuns, while one vessel can belong to multiple Packages.

3.3.3. Labs

Lab 24: Change properties at different object levels.

Lab 25: Read Case Property of Vessel.

Lab 26: Read Flow Direction of PipeRun

Lab 27: Access Piping Point.

Lab 28: Access Signal Point.

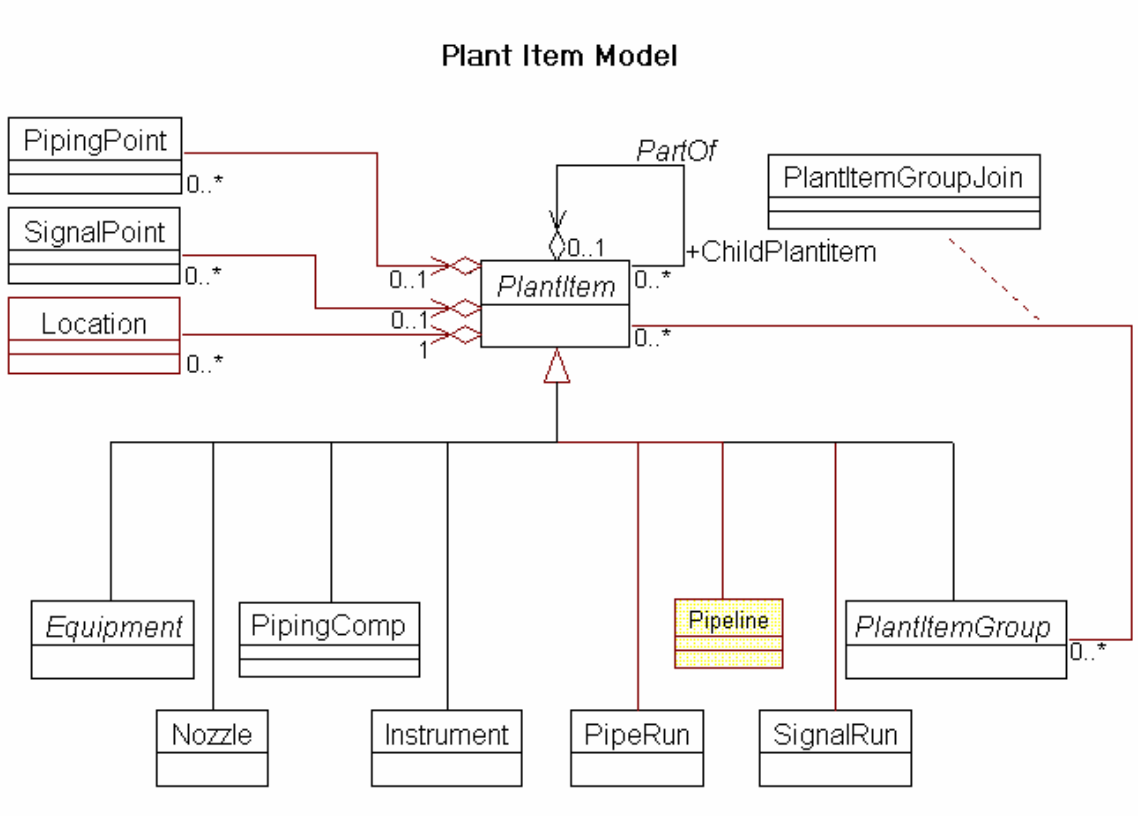
Lab 29: Implied Items.

Lab 30: Part of PlantItem relationships.

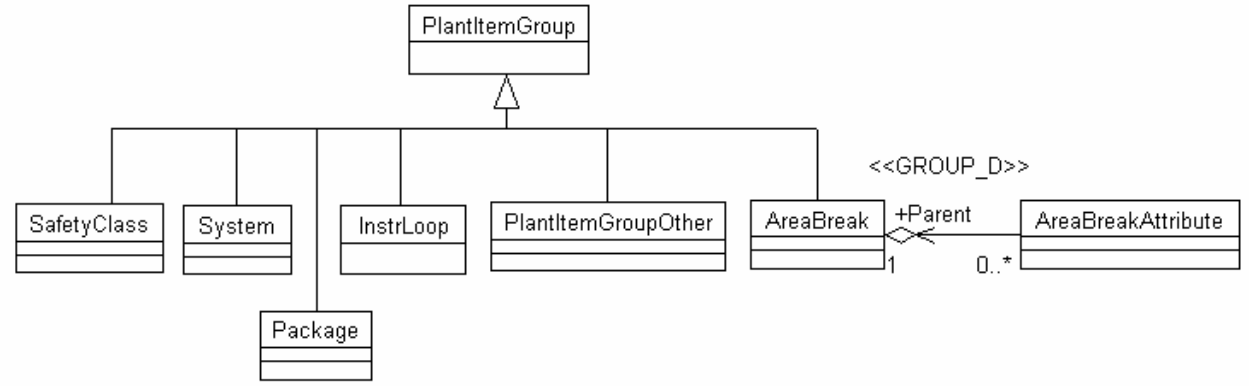
Lab 31: Access Instrument Loop.

Lab 32: LoadInstruments.

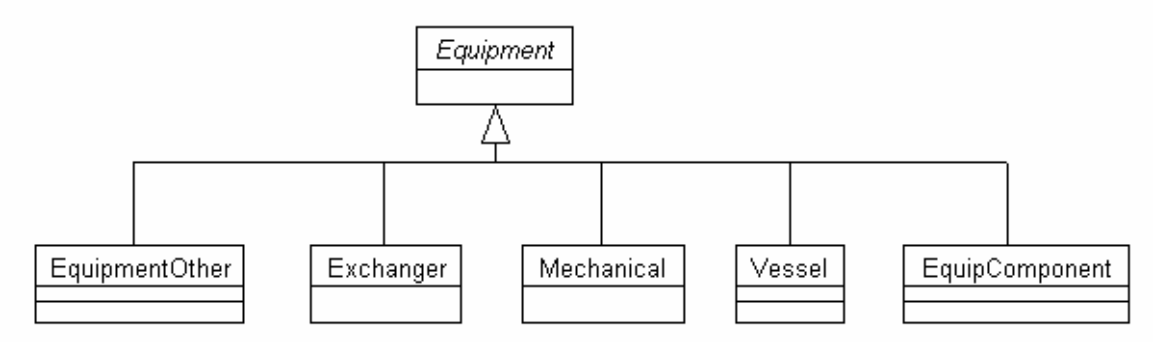
Optional Lab 10: Find Implied Items and Modify Their Property



Model Data: PlantItem



Model Data: PlantItemGroup



Model Data: Equipment

3.4. *Logical Connectivity*

3.4.1. Nozzle - Equipment

A Nozzle is always owned by an Equipment object.

3.4.2.PipingComp – InlineComp

Every PipingComp has an associated InlineComp. The PipingComp owns its associated InlineComp. However, only the InlineComp has the information about the PipeRun that the PipingComp resides on.

3.4.3.Instrument – InlineComp

An Instrument can own an InlineComp if it is an Inline instrument. In this case, the Instrument owns its associated InlineComp. The InlineComp maintains the relationship to the PipeRun that the Instrument is on.

3.4.4. Instrument – PipeRun

When a PipeRun with PiperunClass equals to Instrument is connected with an off-line instrument, SP_PipeRunID in T_Instrument table will be populated with SP_ID of the PipeRun.

3.4.5. Instrument – SignalRun

This relationship is not implemented in the software.

3.4.6.Labs

Lab 33: Identify Nozzle and Equipment.

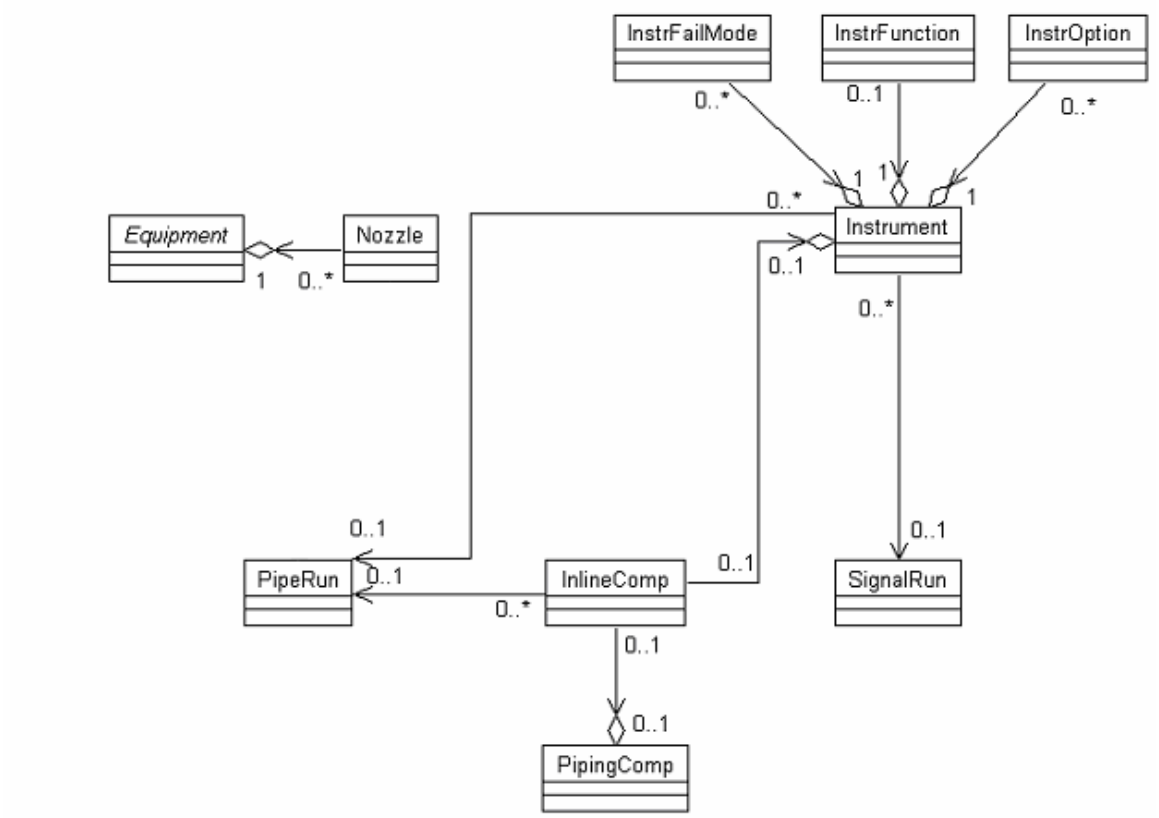
Lab 34: PipingComp and InlineComp.

Lab 35: Instrument and InlineComp

Lab 36: Offline Instrument and SignalRun.

Lab 37: Access Instrument Functions.

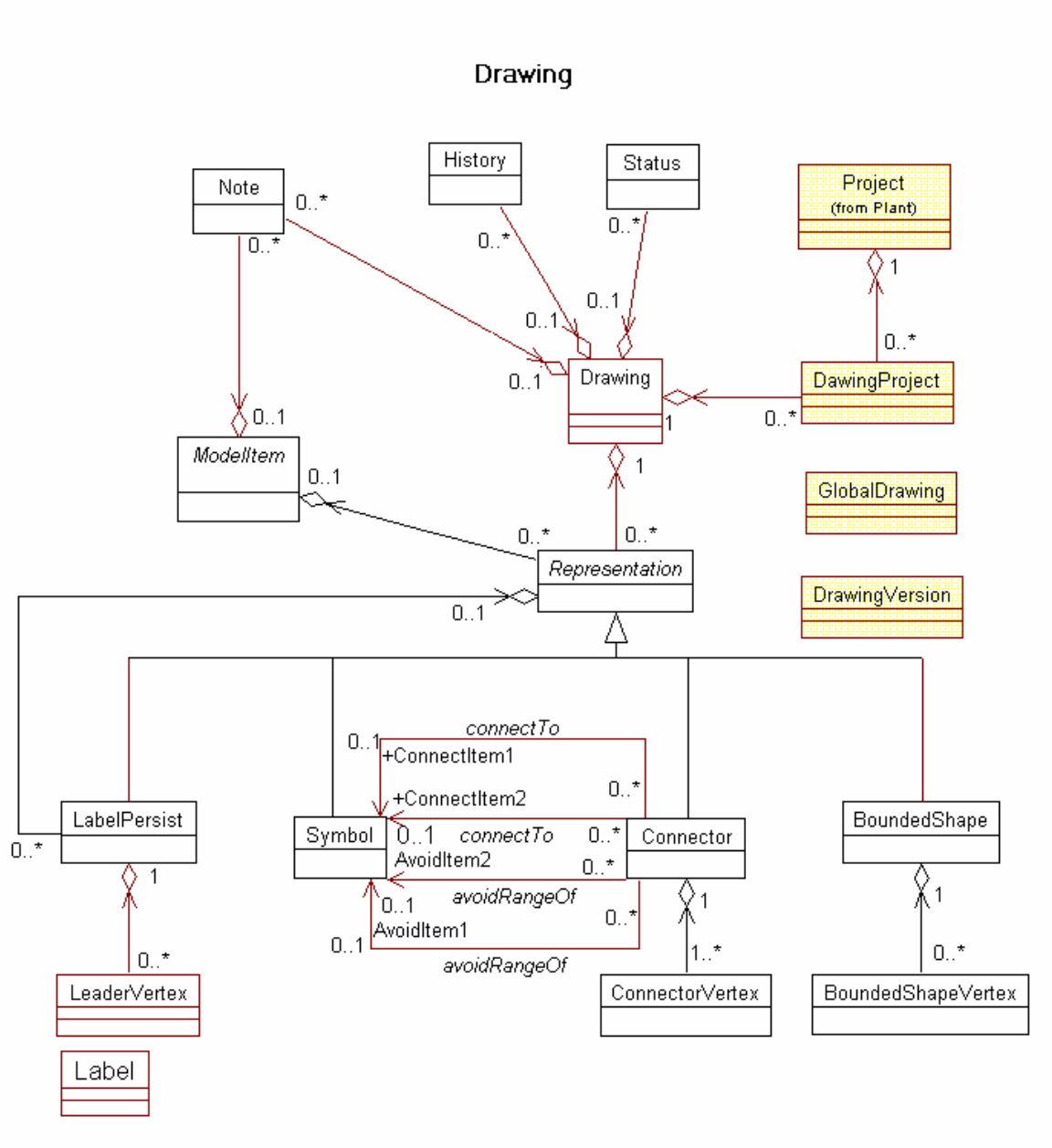
Optional Lab 11: Count Nozzles on a Vessel



Model Data: Connectivity

4. DRAWING DATA

The Drawing Data Model maintains the graphical representation data. It is centered around the Drawing and the Representation objects.



Drawing Data: Package Overview

4.1. *Drawing*

Drawing object can own one or more History, Status and Note. Currently the software creates History records of HistoryType ‘Creation’ and ‘Last Modification’. The user should not modify any of the History records maintained by the system nor create new ones of these types.

4.2. *Representation*

A Representation belongs to a Drawing or Stockpile. Most representations also belong to some ModelItem. There are four types of Representations, namely LabelPersist, Symbol, Connector, and BoundedShape.

4.3. *LabelPersist*

A LabelPersist representation is created every time a label is placed on a drawing. This is the only representation that does not belong to a ModelItem. The LabelPersist representation can have a RepresentationType = 'Label'.

A LabelPersist not only is a Representation, it is also 'Related To' another Representation. This latter representation belongs to the ModelItem on which the label is placed.

For labels on Symbol, the label is labeling the Symbol. However, for labels on PipeRun, actually the label is labeling the Connector.

4.3.1. Labs

Lab 38: Identify connectors of a Piperun.

Lab 39: Find File Name of a Symbol.

Lab 40: Find X, Y Coordinates of Symbol.

Lab 41: Find X, Y Coordinates of Piperun.

Lab 42: Find Labels of a Symbol.

Lab 43: Find Parent Representation of a Label.

Lab 44: Find Parent Drawing of a Symbol.

Lab 45: Find Active Drawing and PlantItems in it.

Lab 46: Filter for Items In Plant Stockpile.

Optional Lab 12: Search Items Active Drawing Stockpile

4.4. *Symbol*

A Symbol representation is created every time an item is placed on a drawing. Every PlantItem, except Pipeline and SignalLine, has an associated Symbol representation. In addition, ModelItems such as OPCs and ItemNotes (annotations) also have symbols. When a PipeRun or a SignalRun is placed on a drawing, multiple representations are created. A Symbol representation is one of the representations placed at the creation of a run. A Symbol representation can have the following RepresentationTypes, namely Gap, OPC, Symbol, and Branch. 'Gap' stands for a gapping symbol

placed at the junction of crossing runs. An 'OPC' type representation is placed for OPC ModelItems. Typically, most other items have a Symbol representation of RepresentationType, 'Symbol'. A branch point on a PipeRun or a SignalRun produces a Symbol of RepresentationType, 'Branch'. The 'Branch' symbol behaves just like a regular symbol.

4.4.1.Connect1Connectors

A collection of connectors that are connected to this current Symbol representation by their End1 ends.

4.4.2.Connect2Connectors

A collection of connectors that are connected to this current Symbol representation by their End2 ends.

4.4.3. ConnectorVertices

A collection of Connector Vertex that holds graphic information for the Connector, such as X, Y Coordinates of the Connector. Pay special attention, X, Y Coordinates are not necessary always stored in the Connector Vertex, sometimes, depending on how the Connector is drawn, the Symbol connected with the Connector holds such information.

4.5. *Connector*

A Connector representation is created along with a Symbol representation when either a PipeRun or a SignalRun is placed, leading to a total of two representations. If an existing connector is split by placing an inline component on the run, then the original connector is deleted and replaced by two connectors, one on each side of the inline component. If the Piperun or SignalRun is deleted into the Stockpile, then all the Connector representations are deleted.

Each connector has two end points, namely the End1 and End2. The ends are named 1 and 2 in the direction that the connector was drawn. Each connector can be attached to a symbol on each of the two ends.

4.5.1.ConnectItem1SymbolID and ConnectItem1SymbolObject

This is the symbol representation to which the End1 of this current connector is attached.

4.5.2.ConnectItem2SymbolID and ConnectItem2SymbolObject

This is the symbol representation to which the End2 of this current connector is attached.

4.6. *BoundedShape*

A BoundedShape representation is created when placing an AreaBreak or Revision Cloud.

4.7. *Labs*

Lab 47: Identify items connected to a PipeRun.

Lab 48: Identify the PipeRun associated with the PipingComp.

Lab 49: Navigate down a piperun to a vessel.

Lab 50: Navigate through a branch point on a piperun.

Lab 51: Navigate through OPC

Optional Lab 13: Navigate from Off-line Instrument to Process PipeRun.

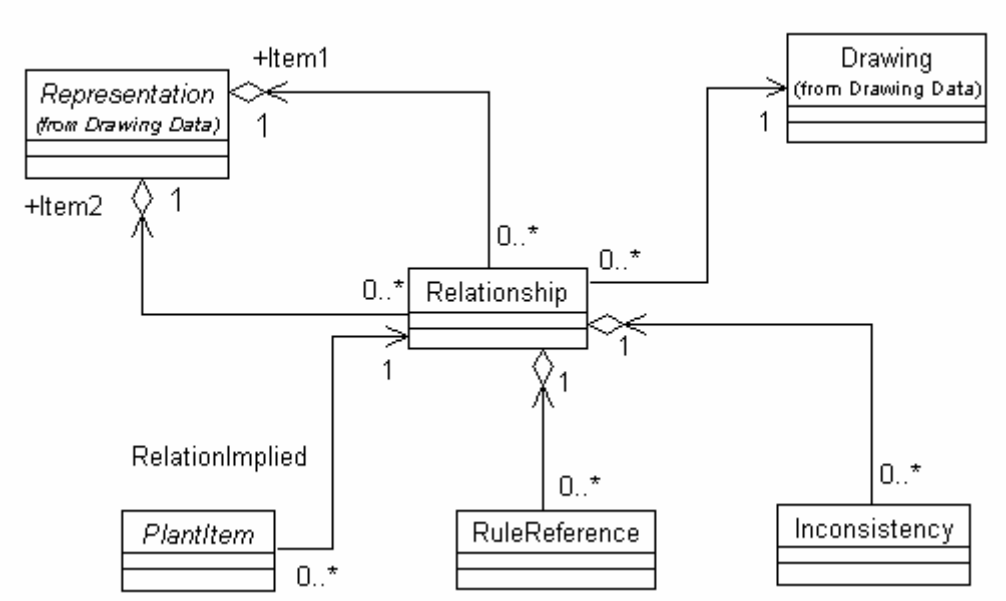
Optional Lab 14: Find OPC and From/To.

5. ADDITIONAL DATA MODEL

5.1. *Relationship*

A Relationship is created automatically between representations when

(1) Two graphics are connected. Example: PipeRun connected to Valve, Nozzle on a Vessel. (2) When a symbol is placed that has piping points or signal points.



Relationship Model

5.2. *Labs*

Lab 52: Access Relationship from Representation

Lab 53: Access Inconsistency

Lab 54: Access RuleReference

5.3. *PlantGroup*

PlantGroup is a superclass with eight subclasses, namely Site, Plant, Unit, BusinessSector, Level, PlantSystem, SubSystem and Area. These are designated by PlantGroupTypes enumerated list in the DataDictionary. Unlike all other objects in the SPPID data model, new subclasses can be added to PlantGroups. These classes are persisted as tables with a 'SPM' prefix after the prefix 'T_'. For example, a user-defined PlantGroup SubArea corresponding table is T_SPMSubArea. These objects can be obtained as generic LMAItems. For instance, if SubArea is a user-defined PlantGroup that was included into a hierarchy and used in a plant structure, then the following code can be used to access the SubArea that has an ID="SP_ID":

```
Dim objItem As LMAItem
Set objItem = objDatasource.GetItem("SPMSubArea", "SP_ID")
```

The attributes of the SubArea will need to be obtained through the Attributes collection because they are user-defined.

5.3.1. ParentID and ParentType

ParentID returns the ID of the Parent PlantGroup of the current PlantGroup. It returns a negative 1 (-1) if it is the top-most item in the hierarchy. The ParentType is a number describing the type of the Parent PlantGroup.

5.3.2. PlantGroupType

It is an enumerated list describing the type of the PlantGroup.

5.3.3. PlantItems

A collection of PlantItems that have been associated with this PlantGroup. One example is when an Equipment Vessel is associated with a Unit. The Equipment would then appear in the Unit's PlantItems collection.

5.3.4. Drawings

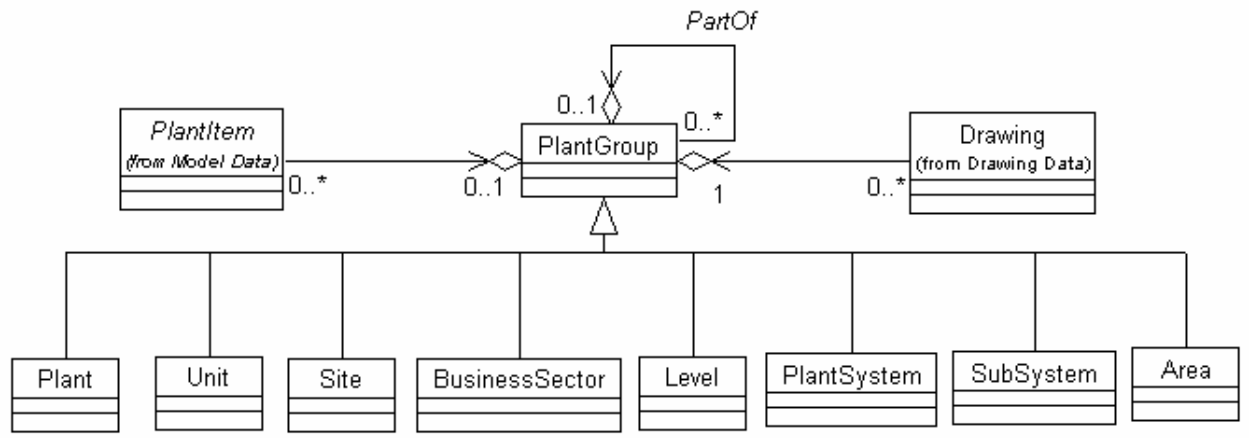
A collection of Drawings that are directly under the current PlantGroup.

5.3.5.Labs

Lab 55: Access PlantGroup from PlantItem.

Lab 56: Access PlantGroup from Drawing.

Lab 57: Access Customized PlantGroup.



Plant Hierarchy

5.4. *Workshare*

Workshare may be set up to share SPPID data among multiple locations or to have a Task/Master Plant. Workshare entities are created in PLANT schema. LLAMA is aware of the workshare. For example, if user tries to modify a property of an item that read-only because of workshare, the modification will not be persisted to database. However, there is no public interface exposed in LLAMA that user can check such as ownership of a drawing, or user's access right.

5.4.1.WSSite

Every plant, by default, is a workshare site. User can create multiple satellite workshare sites in the plant.

5.4.2.DrawingSite

This is a join table between T_WSSite table and T_Drawing table. Through this table, drawing knows who is it owner workshare site, and workshare site knows how many drawings belongs to it.

5.4.3.ActiveWSSite

A plant may have multiple workshare sites, but it can have only one active workshare site.

5.4.4.PlantItemGroup

A table contains plant item group information that related to a workshare site.

5.4.5.OPC

A table contains OPC information that related to a workshare site.

5.4.6.PlantGroup

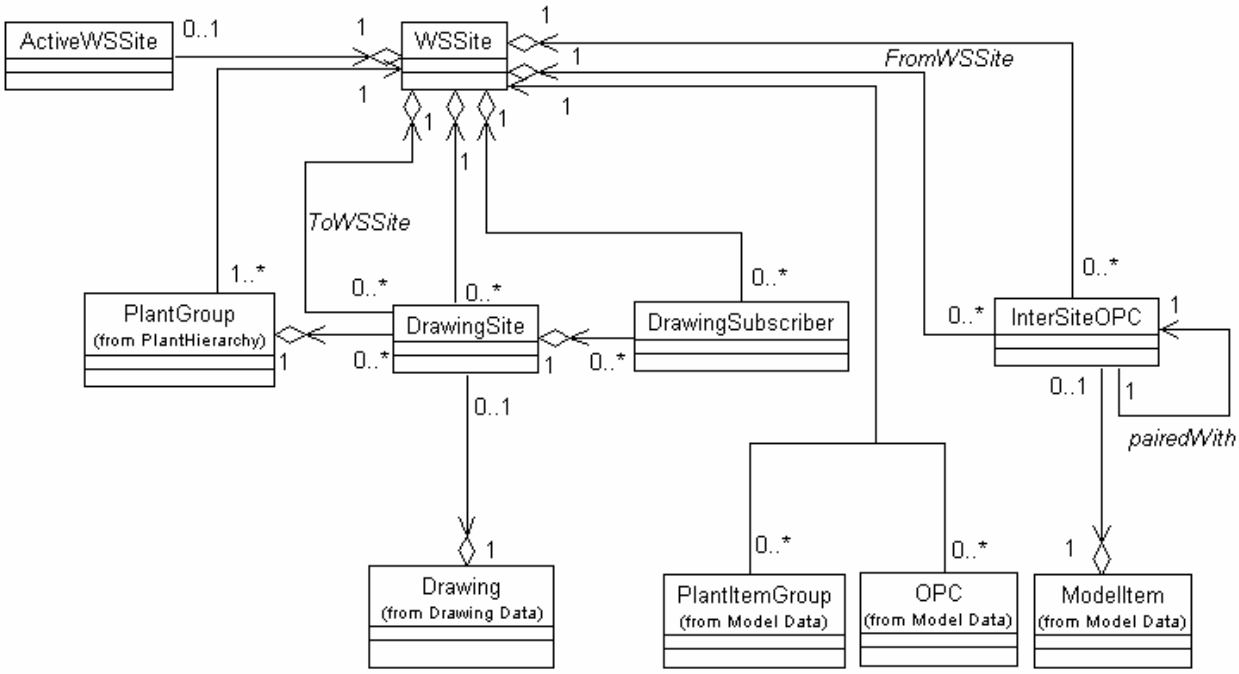
T_PlantGroup has a point that points back to the workshare site.

5.4.7.Labs

Lab 58: Access Workshare Site

Lab 59: Access DrawingSite

Lab 60: Workshare Awareness in LLAMA



Workshare Model

5.5. *As-Build/Project*

5.5.1.Project

Every plant, by default, is The Plant. Then, User can create multiple projects in the plant.

5.5.2. ActiveProject

The current in-use project is the active project. User can access active project through LMADatasource.GetActiveProject function. It can be The Plant or projects.

5.5.3. Project Status

Project status is a select list data with following lists: Active, Completed, Merged, Finished, Cancelled, Terminated, None, Deleted. The Plant status is None. Projects can have different statuses.

5.5.4. Claim Status

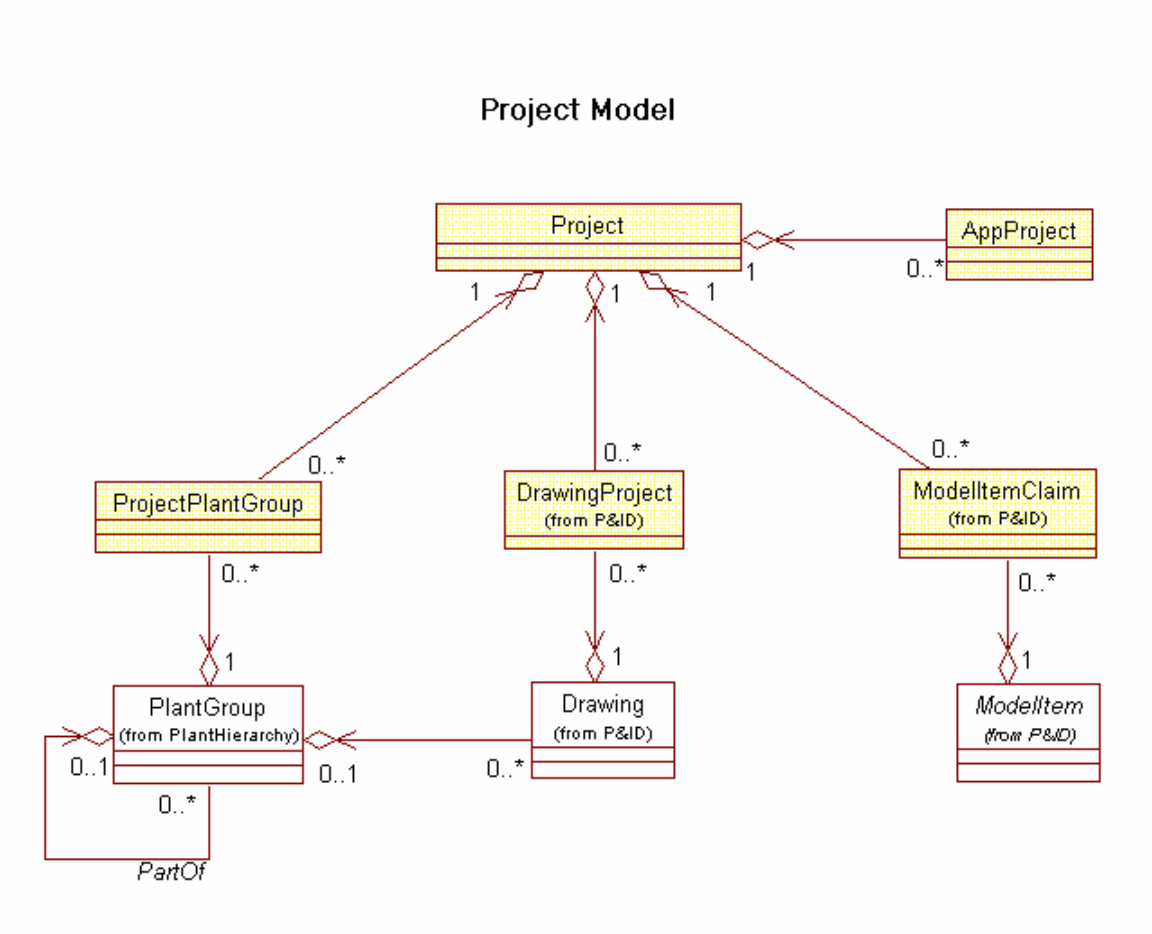
In As-Build/Project environment, an item can have different claim status: Not Claimed, Claimed by active project, and Claimed by Others.

5.5.5. Labs

Lab 61: Access Active Project

Lab 62: Access Plant from Project

Lab 63: Access Claim Status of Items



Project Model

5.6. *T_OptionSetting Labs*

Lab 64: Access OptionSettings

Optional Lab 15: Copy a Customized Property of Unit to PipeRun

5.7. *Special Issues*

5.7.1. Model Item always returns in its concrete level

Since V4, all model item object will be obtained in its concrete level. For example, even retrieved as n LMEquipment, its ItemType is “Vessel”. In addition, the object will have all ItemAttributions collection as its concrete object.

CHAPTER 8: PLACEMENT AUTOMATION

1. OBJECTIVES

In this chapter, you will learn

◊ an overview of the Placement Automation library

2. PLAICE LIBRARY

2.1. *Create an Item in the Stockpile*

Function PIDCreateItem(DefinitionFile As String, [DrawingID="0"]) As LMAItem

DefinitionFile: the path and name of symbol file
DrawingID: SP_ID of drawing, the item will be created in its stockpile. If user leaves it blank, by default, value is 0, which means plant stockpile. If you user specify a valid drawing SP_ID, the item will be created in that drawing's stockpile.

2.2. *Place an Symbol on the Drawing*

Function PIDPlaceSymbol(DefinitionFile As String, X As Double, Y As Double, [Mirror], [Rotation], [ExistingItem As LMAItem], [TargetItem]) As LMSymbol

DefinitionFile: the path and name of symbol file
X: x-coordinate of where the symbol locates on Drawing, unit in METER
Y: y-coordinate of where the symbol locates on Drawing, unit in METER

[Mirror]: True or False
[Rotation]: 1.57=90 Degree
[ExistingItem]: Item as LMAItem in stockpile/drawing to be placed on Drawing
[TargetItem]: Related item as LMRepresentation of the symbol to be placed, for example, when place a nozzle symbol, a Equipment item must be specified as TargetItem. (When TargetItem is specified, given X, Y coordinates may be ignored, software will locate a new X, Y coordinates for the symbol to be located on the edge of TargetItem, if PIDSnapToTarget is set to TRUE)

2.3. *Place Labels*

Function PIDPlaceLabel(DefinitionFile As String, Points() As Double, [Mirror], [Rotation], [LabeledItem As LMRepresentation], [IsLeaderVisible As Boolean = False]) As LMLabelPersist

DefinitionFile: the path and name of label file
Points(): a dynamic array, specifies the X, Y coordinates for Label to be placed on drawing
[Mirror]: True or False
[Rotation]: 1.57=90 Degree
[LabeledItem]: LMRepresentation of targetitem on which the label to be placed

[IsLeaderVisible]: specifies the Leader of the label to be placed is visible or not

2.3.1.Labs

Lab 65: Create a Vessel and place into StockPile.

Lab 66: Place a Vessel on a Drawing.

Lab 67: Place Nozzles and Trays on a Vessel.

Lab 68: Place Labels on a Vessel.

Lab 69: Place OPC

Lab 70: Place OPC from StockPile

2.4. *Place Piperun*

Function PIDPlaceRun(StockpileItem As LMAItem, Inputs As PlaceRunInputs) As LMConnector

StockpileItem: Piperun placed in the stockpile as LMAItem

Inputs: Collection of input parameters as PlaceRunInputs

PlaceRunInputs.AddPoint(X As Double, Y As Double)

PlaceRunInputs.AddLocatedTarget(X As Double, Y As Double, [Diagonal As Boolean = False])

PlaceRunInputs.AddConnectorTarget(TargetItem As LMConnector, X As Double, Y As Double, [Diagonal As Boolean = False])

PlaceRunInputs.AddSymbolTarget(TargetItem As LMSymbol, X As Double, Y As Double, [Diagonal As Boolean = False])

2.5. *Auto Join Piperuns*

Sub PIDAutoJoin(RunItem As LMAItem, AutoJoinEnd As AutoJoinEndConstants, SurvivorItem As LMAItem)

RunItem: Exiting Piperun as LMAItem to seek AutoJoin other piperuns

AutoJoinEnd: Enum data as values are: autojoin_both, autojoin_start, autojoin_end, and autojoin_none.

SurvivorItem: When two piperuns auto joined, one piperun will disappear while the other will survive, the survived one is the SurvivorItem.

2.6. *Place Bounded Shapes*

Function PIDPlaceBoundedShape(DefinitionFile As String, Points() As Double, [ExistingItem As LMBoundedShape]) As LMBoundedShape

DefinitionFile: the path and name of symbol file

Points: a dynamic array, specifies the X, Y coordinates for BoundedShape to be placed on drawing

ExistingItem: Item in stockpile to be placed on Drawing

2.7. *Place Assemblies*

Function PIDPlaceAssembly(AssemblyFile As String, X As Double, Y As Double) As LMAItems

AssemblyFile: the path and name of symbol file

X: x-coordinate of where the symbol locates on Drawing, unit in METER

Y: y-coordinate of where the symbol locates on Drawing, unit in METER

2.8. *Place Gaps*

Function PIDPlaceGap(DefinitionFile As String, GapX As Double, GapY As Double, ParamLeft As Double, ParamRight As Double, [objLMConnector As LMConnector], [RotationAngle], [ExistingRun As LMAItem]) As LMSymbol

DefinitionFile: the path and name of symbol file
GapX: x-coordinate of where the Gap locates on Drawing, unit in METER

GapY: y-coordinate of where the Gap locates on Drawing, unit in METER

ParamLeft: length of Gap to the Left, unit in METER

ParamRight: length of Gap to the Right, unit in METER

[ObjLMConnector]: target PipeRun on which the Gap is placed

[RatationAngle]: 1.57=90 Degree
[ExistingRun]**: not working now

2.8.1. Labs

Lab 71: Using PIDPlaceRun to Place Piperun

Lab 72: Using PIDAutoJoin to Auto Join Two Piperuns

Lab 73: Place Gap.

Lab 74: Place BoundedShape

Lab 75: Place Assembly.

2.9. *Remove a Symbol from the Drawing*

Function PIDRemovePlacement(Representation As LMRepresentation) As Boolean

Representation: LMRepresentation of the symbol on Drawing to be removed into stockpile

2.10. *Delete an Item from Model*

Function PIDDeleteItem(Item As LMAItem) As Boolean

Item: LMAItem of the symbol (on Drawing) or item (in stockpile) to be delete from Model

2.11. *Replace Symbol*

Function PIDReplaceSymbol(NewSymbolFileName As String, ExistingSymbol As LMSymbol) As LMSymbol

NewSymbolFileName: the path and name of symbol file used to replace the existing symbol

ExistingSymbol: LMSymbol of symbol on the Drawing to be replaced

2.12. *Replace Label*

Function PIDReplaceLabel(NewSymbolFileName As String, ExistingLabel As LMLabelPersist) As LMLabelPersist

NewSymbolFileName: the path and name of label file used to replace the existing Label

ExistingLabel: LMLabelPersist of Label on the Drawing to be replaced

2.13. *Replace OPC*

Function PIDReplaceOPC(NewSymbolFileName As String, objExistingOPC As LMSymbol) As LMSymbol **Not working now

NewSymbolFileName: the path and name of OPC used to replace the exising OPC

objExistingOPC: LMSymbol of OPC on the Drawing to be replaced

(However, pairOPC is not replaced)

2.14. *Apply Parameters to Parametric Symbols*

Sub PIDApplyParameters(Representation As LMRepresentation, Names() As String, Values() As String)

Representation: LMRepresentation of parametric symbol on Drawing, whose parameters are to be modified

Names(): a dynamic array, specifies the names of parameter, for example: TOP, RIGHT

Values(): a dynamic array, specifies the values of parameter

2.15. *Locate Connect Point*

Function PIDConnectPointLocation(Symbol As LMSymbol, ConnectPointNumber As Long, X As Double, Y As Double) As Boolean

Symbol(): LMSymbol of the symbol to be located of connect point

ConnectPointNumber: Index number of the connect point on a symbol

X: x-coordinate of the connect point on drawing, unit in METER

Y: y-coordinate of the connect point on drawing, unit in METER

2.16. *Datasource*

Property PIDDataSource As LMADataSource

Property returns a LMADataSource object

2.17. *PIDSnapToTarget*

Property PIDSnapToTarget As Boolean

Property specifies item to be placed snap to targetitem or not, TRUE by default

2.18. *SetCopyPropertiesFlag*

Function PIDSetCopyPropertiesFlag(bDoCopy As Boolean) As Boolean

Property specifies the properties to be copied or not as defined in the Rule across the Items that are connected.

2.18.1. Labs

Lab 76: Remove Vessel.

Lab 77: Delete Vessel.

Lab 78: Replace Symbol.

Lab 79: Replace Label.

Lab 80: Replace OPC.

Lab 81: Modify Parametric Symbol.

Lab 82: Using PIDConnectPointLocation to Locate X, Y Coordinates.

Lab 83: Create Instrument Loop.

Lab 84: Find and Replace Label.

2.19. *Datasource*

Property PIDDataSource As LMADataSource

Property returns a LMADataSource object

2.20. *Place Connectors*

Function PIDPlaceConnector(DefinitionFile As String, Points() As Double, [OrthogonalStart As Boolean = True], [OrthogonalEnd As Boolean = True], [ExistingItem As LMAItem], [OriginItem], [DestinationItem], [OriginIndex As Long = 1], [DestinationIndex As Long = 1], [MinSegLength As Double = 0.001], [RangeClearance As Double = 0.004]) As LMConnector

DefinitionFile: the path and name of symbol file
Points(): a dynamic array, specifies the X, Y coordinates for Connector to be placed on drawing
[OrthogonalStart]: specifies if Orthogonal to OriginItem
[OrthogonalEnd]: specifies if Orthogonal to DestinationItem

[ExistingItem]: LMAItem of existing piperun in the same drawing
[OriginalItem]: LMRepresentation of original item where connector starts
[DestinationItem]: LMRepresentation of Destination item where connector ends
[OriginIndex]**: not working now
[DestinationIndex]**: not working now
[MinSegLength]: specifies the minimum segment length to drawing the connector
[RangeClearance]: specifies the avoid range when place the connector
(when specifies the OriginalItem and DestinationItem, the first point's and last point's X, Y coordinates will be ignored)
(When placing a connector to a symbol, how to control which connect point on symbol the connector should be connected? Instead of using OriginIndex and DestinationIndex, specifying the X, Y coordinate for connector point you wish to connect with in Points(), which will make the connector to connect the connect point of the symbol)

3. MISC.

When placing a piping component on a piperun connector, the original connector is given an ItemStatus of 'Delete – Pending' or ItemStatusIndex = 4 and two new connectors are created. When placing the next item on the connector, you cannot use the reference to the original connector because it is no longer valid. We have to find the two new connectors and then use their references. This can be achieved by looking at the symbol placed and finding the item1connectors or item2connectors.

When place a Gap on a piperun, the symbol does not break the original connector either. It creates two more new connects and sits on the top of the piperun. When user select not to see the Gap, the Gap along with two connectors it created will go to hidden layer and original connector is showing.

4. LABS

Optional Lab 16: Label Find and Replace Utility

CHAPTER 9: USING PIDAUTOMATION TO CREATE, OPEN AND CLOSE DRAWINGS

1. OBJECTIVES

In this chapter, you will learn

◊ Using PIDAutomation to create, open and close drawings.

2. THREE IMPORTANT FUNCTIONS

2.1. *PIDAutomation.Application.Drawings.Add*

Function Add(PlantGroupName As String, TemplateFileName As String, DrawingNumber As String, DrawingName As String, [DocumentTypeValue], [DocumentCategoryValue], [bVisible As Boolean = True]) As Drawing

PlantGroupName: Name of PlantGroup that is just above drawing in Plant Structure Hierarchy

TemplateFileName: Full path and name of drawing template file

DrawingNumber: Drawing number of the drawing to be created

DrawingName: Drawing name of the drawing to be created

DocumentTypeValue: Optional argument, select list value for document type, default value is 631 stands for P&Ids

DocumentCategoryValue: Optional argument, select list value for document category, default value is 6 stands for Piping Documents

BVisible: Optional argument, Boolean value as True or False for the visibility of drawing when opening, default as True

2.2. *PIDAutomation.Application.Drawings.OpenDrawing*

Function OpenDrawing(DrawingName As String, [Visible = True]) As Drawing

DrawingName: Pure name of drawing

Visible: Optional argument, Boolean value as True or False for the visibility of drawing when opening, default as True

2.3. *PIDAutomation.Drawing.CloseDrawing*

Sub CloseDrawing(SaveFile As Boolean)

SaveFile: Boolean value as True or False to save drawing when closing drawing.

3. LABS

Lab 85: Open and close an Existing Drawing

Lab 86: Create, Open and Close a New Drawing

Lab 87: Comprehensive Automation Lab

Optional Lab 17: Automatically Create New Drawings

CHAPTER 10: CALCULATION/VALIDATION USING ACTIVE-X SERVER COMPONENTS

1. OBJECTIVES

In this chapter, you will learn

◊ the special issues relevant to writing Calculation/Validation routines

2. ILMFOREIGNCALC INTERFACE

The ILMForeignCalc Interface supports four functions that are called by SmartPlant P&ID at specific events. This interface is supplied in the LMForeignCalc library. `IMPLEMENTS ILMFOREIGNCALC` statement incorporates this interface into the Active-X component. The functions are described below:

2.1. *DoCalculate*

◊ Function DoCalculate(DataSource As LMADataSource, Items As LMAItems, PropertyName As String, Value) As Boolean

◊ It is activated only if the ProgID of the class has been entered into the Calculation ID field of the property in the DataDictionary Manager.

◊ SPPID triggers a call to the function when the user clicks on the ellipsis button on the Property Grid next to the relevant property.

2.2. *DoValidateItem*

◊ Function DoValidateItem(DataSource As LMADataSource, Items As LMAItems, Context As ENUM_LMAValidateContext) As Boolean

◊ It is activated only if the ProgID of the class has been entered into the Validation Program field of the Database Item Type in the DataDictionary Manager.

◊ SPPID triggers a call to the function when the user place a new item, copy and paste and existing item, and delete and item to/from drawing

◊ SPPID triggers a call to the function when the user de-selects from the item after changing some property associated with the item.

- ◊ The various contexts for the triggering are listed by the enumerated variable LMForeignCalc.ENUM_LMAValidateContext, which has values of LMAValidateCreate, LMAValidateCopy, LMAValidateDelete, LMAValidateModify and LMAValidateFiltered**.
- ◊ The call is made to this function generally after every relationship has been created and the rules have been executed.

2.3. *DoValidateProperty*

- ◊ Function DoValidateProperty(DataSource As LMADataSource, Items As LMAItems, PropertyName As String, Value) As Boolean
- ◊ It is activated only if the ProgID has been entered into Validation ID field of the property in the DataDictionary Manager.
- ◊ SPPID triggers a call to the function when the user de-selects from the relevant property after having modified it.
- ◊ The call is made to this method before SPPID accepts the user's entry into the memory.

2.4. *DoValidatePropertyNoUI*

- ◊ Sub DoValidatePropertyNoUI(DataSource As LMADataSource, Items As LMAItems, PropertyName As String, Value)
- ◊ It is activated only if the ProgID has been entered into Validation ID field of the property in the DataDictionary Manager.
- ◊ SPPID triggers a call to this function when a rule modifies the property, such as at placement time.

3. OBJECTS PASSED FROM THE MODELER

3.1. *LMADataSource*

This LMADataSource is initialized to the Project associated with the Drawing. A ProjectNumber property will not return anything and it cannot be set to any other Project than the one associated with the Drawing.

3.2. *LMAItems*

The select set of items is passed to the Calculation/Validation routines as a collection of LMAItems.

3.3. *PropertyName*

This is the name as listed in the ItemAttributions table of the DataDictionary.

3.4. *Property Value*

This is a Variant and can contain a NULL value.

4. SPECIAL ISSUES

4.1. *Updating the Property Grid*

In order to update the value seen through the Property Grid, it is necessary to set the value of the 'Value' variable and to return a TRUE for the Boolean return value of the ILMForeignCalc functions. If the return value is FALSE (default), then it is assumed that the operation was not a success and the old values are retained in the property grid.

4.2. *Commit command*

A Commit command must be issued to the object in question if the changes to its properties other than the property being validated are to be made persistent.

4.3. *Automatically Fire Up Validation*

When user uses automation program to update a property, if the property has a validation ProgID in its validation ProgID field, the validation will be automatically called up. For example, when user update the Vessel's prefix to "P" and commit the change, then validation will be fired up to generate a new item tag for Vessel if the property TagPrefix has a validation ProgID in its validation ProgID field.

5. LABS

Lab 88: Create a Calculation Program.

Lab 89: Create a ValidateProperty Program.

Lab 90: Create a ValidateItem Program.

Optional Lab 18: Calculation Validation (1).

Optional Lab 19: Calculation Validation (2).

Optional Lab 20: Property Validation (1).

Optional Lab 21: Property Validation (2).

Optional Lab 22: Item Validation (1).

Optional Lab 23: Item Validation (2).

Optional Lab 24: Item Validation (3).

CHAPTER 11: USING LMAUTOMATIONUTIL TO GET FROM/TO INFORMATION

1. OBJECTIVES

In this chapter, you will learn

◊ the special issues relevant to use LMAutomationUtil to get from/to information for Piperun.

2. IMPORTANT METHODS

2.1. *RunsNavigation*

Sub RunsNavigation(objRun As LMPipeRun, FromCollection As Collection, ToCollection As Collection, [InDeterminateCol as Collection)

objRun: LMPipeRun of the Piperun object to report of its from/to

FromCollection: Collection of items that are returned as FROM Items

ToCollection: Collection of items that are returned as TO Items

InDeterminateCol: Collection of items that can not be determined ad FROM or TO items

2.2. *RunsNavigationAll*

Sub RunsNavigationAll(objDS As LMADataSource, objRuns As String, BuildInDeterminates As Boolean, OutDataCollection As Collection)

objDS: Current LMADataSource

objRuns: A string value with all SP_IDs of PipeRuns in format as 'SP_ID','SP_ID',…

BuildInDeterminates: A Boolean value, if it is TRUE, a FROM/TO item, such as PipeRun, will be reported even no flow direction is defined on that PipeRun

OutDataCollection: Collection of Collection that includes the FROM/TO items

Sub RunsNavigationAll(objDS As LMADataSource, objRuns As String, BuildInDeterminates As Boolean, OutDataCollection As Collection)

Each item, which is object as LMAutomationUtil.ToFromItem, in OutDataCollection is a collection of all from/to item related to a Piperun.

LMAutomationUtil.ToFromItem has following functions/methods/properties:

BucketType: data type as LONG, 1 means From, 2 means To, 3 means BiDir

Connected_FlowDirection: data type as LONG, code list index value of FlowDirection of the piperun that is reported as from/to item

Connected_ID: data type as string, from/to item's SP_ID

Connected_ItemTag: data type as string, from/to item's ItemTag

Connected_OPC_DrawingNumber: data type as string, if From/to item is OPC, and this OPC has pair OPC on a drawing, then the Drawing Number of that drawing where its pair is sitting on.

Equipment_ItemTag: data type as string, if from/to item is NOZZLE, then the itemtag of the Equipment the Nozzle is on.

Equipment_SP_ID: data type as string, if from/to item is NOZZLE, then the SP_ID of the Equipment the Nozzle is on.

Init: internal method

ItemType: data type as string, the ItemType of the from/to item

PipeRun_FlowDirection: data type as LONG, code list index value of FlowDirection of the piperun that is running from/to report

PipeRun_ID: data type as string, SP_ID of FlowDirection of the piperun that is running from/to report

3. LOGIC OF FROM/TO MACRO

To run from/to for a Piperun (Piperun A), flow direction has to be defined for the Piperun (Piperun B). Then, by default, there are three item types can be reported as from/to items: OPC, Nozzle and Piperun (Piperun B). For the Piperun (Piperun B) to be reported as from/to item, flow direction must be defined on it also. However, if there the parameter “BuildInDeterminates” is set to TRUE in function RunsNavigationAll, then the Piperun (Piperun B) will be reported as from/to item even without flow direction defined, it will be reported with TFItem.bucketType=4, to print this information out, user needs to modify the from/to code a little bit to print it out.

In addition, pipingcomp or inline instrument can be reported as from/to items also, but there are procedures to turn it on. Please follow steps as followings:

1) add new property in Inline Component table
Open up Data Dictionary using the Data Dictionary Manager
Click on Database Items
Choose In Line Component
right click on the attributes of inline component and choose Add Property
In the form

1. Name = EndComponent
2. Display Name = End Component
3. Data Type = Select List
4. Select List = Boolean Values
5. Format Value = Variable Length
6. Default Value = False
7. Maximum Length = 5
8. Display To User = Yes
9. Use for Filtering = No
10. Category = {Identification}

Click OK and save the changes

2) set new property 'EndComponent' to True for any special PipingComp or Inline Instrument to be True, then the special PipingComp or Inline Instrument will be returned in piperunNode Collection as Piperun, Nozzle or OPC.

4. LABS

Optional Lab 29: Improvement of from/to macro