Smart P&ID 2019

Automation Programming Course Guide

Version 2019 **July 2020** **D PID2-TP-100029A**

Copyright

U.S. Government Restricted Rights Legend

Intergraph Corporation
305 Intergraph Way
Madison, AL 35758

Documentation

Documentation shall mean, whether in electronic or printed form, User's Guides, Installation Guides, Reference Guides, Administrator's Guides, Customization Guides, Programmer's Guides, Configuration Guides and Help Guides delivered with a particular software product.

Other Documentation

Other Documentation shall mean, whether in electronic or printed form and delivered with software or on Intergraph Smart Support, SharePoint, or box.net, any documentation related to work processes, workflows, and best practices that is provided by Intergraph as guidance for using a software product.

Terms of Use

Use of a software product and Documentation is subject to the Software License Agreement ("SLA") delivered with the software product unless the Licensee has a valid signed license for this software product with Intergraph Corporation. If the Licensee has a valid signed license for this software product with Intergraph Corporation, the valid signed license shall take precedence and govern the use of this software product and Documentation. Subject to the terms contained within the applicable license agreement, Intergraph Corporation gives Licensee permission to print a reasonable number of copies of the Documentation as defined in the applicable license agreement and delivered with the software product for Licensee's internal, non-commercial use. The Documentation may not be printed for resale or redistribution.

For use of Documentation or Other Documentation where end user does not receive a SLA or does not have a valid license agreement with Intergraph, Intergraph grants the Licensee a non-exclusive license to use the Documentation or Other Documentation for Licensee's internal non-commercial use. Intergraph Corporation gives Licensee permission to print a reasonable number of copies of Other Documentation for Licensee's internal, non-commercial use. The Other Documentation may not be printed for resale or redistribution. This license contained in this subsection b) may be terminated at any time and for any reason by Intergraph Corporation by giving written notice to Licensee.

Disclaimer of Warranties

Except for any express warranties as may be stated in the SLA or separate license or separate terms and conditions, Intergraph Corporation disclaims any and all express or implied warranties including, but not limited to the implied warranties of merchantability and fitness for a particular purpose and nothing stated in, or implied by, this document or its contents shall be considered or deemed a modification or amendment of such disclaimer. Intergraph believes the information in this publication is accurate as of its publication date.

The information and the software discussed in this document are subject to change without notice and are subject to applicable technical product descriptions. Intergraph Corporation is not responsible for any error that may appear in this document.

The software, Documentation and Other Documentation discussed in this document are furnished under a license and may be used or copied only in accordance with the terms of this license. THE USER OF THE SOFTWARE IS EXPECTED TO MAKE THE FINAL EVALUATION AS TO THE USEFULNESS OF THE SOFTWARE IN HIS OWN ENVIRONMENT.

Intergraph is not responsible for the accuracy of delivered data including, but not limited to, catalog, reference and symbol data. Users should verify for themselves that the data is accurate and suitable for their project work.

Limitation of Damages

IN NO EVENT WILL INTERGRAPH CORPORATION BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL INCIDENTAL, SPECIAL, OR PUNITIVE DAMAGES, INCLUDING BUT NOT LIMITED TO, LOSS OF USE OR PRODUCTION, LOSS OF REVENUE OR PROFIT, LOSS OF DATA, OR CLAIMS OF THIRD PARTIES, EVEN IF INTERGRAPH CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

UNDER NO CIRCUMSTANCES SHALL INTERGRAPH CORPORATION'S LIABILITY EXCEED THE AMOUNT THAT INTERGRAPH CORPORATION HAS BEEN PAID BY LICENSEE UNDER THIS AGREEMENT AT THE TIME THE CLAIM IS MADE. EXCEPT WHERE PROHIBITED BY APPLICABLE LAW, NO CLAIM, REGARDLESS OF FORM, ARISING OUT OF OR IN CONNECTION WITH THE SUBJECT MATTER OF THIS DOCUMENT MAY BE BROUGHT BY LICENSEE MORE THAN TWO (2) YEARS AFTER THE EVENT GIVING RISE TO THE CAUSE OF ACTION HAS OCCURRED.

IF UNDER THE LAW RULED APPLICABLE ANY PART OF THIS SECTION IS INVALID, THEN INTERGRAPH LIMITS ITS LIABILITY TO THE MAXIMUM EXTENT ALLOWED BY SAID LAW.

Export Controls

Intergraph Corporation's commercial-off-the-shelf software products, customized software and/or third-party software, including any technical data related thereto ("Technical Data"), obtained from Intergraph Corporation, its subsidiaries or distributors, is subject to the export control laws and regulations of the United States of America. Diversion contrary to U.S. law is prohibited. To the extent prohibited by United States or other applicable laws, Intergraph Corporation software products, customized software, Technical Data, and/or third-party software, or any derivatives thereof, obtained from Intergraph Corporation, its subsidiaries or distributors must not be exported or re-exported, directly or indirectly (including via remote access) under the following circumstances:

a. To Cuba, Iran, North Korea, the Crimean region of Ukraine, or Syria, or any national of these countries or territories.

To any person or entity listed on any United States government denial list, including, but not limited to, the United States Department of Commerce Denied Persons, Entities, and Unverified Lists, the United States Department of Treasury Specially Designated Nationals List, and the United States Department of State Debarred List (https://build.export.gov/main/ecr/eg_main_023148).

To any entity when Customer knows, or has reason to know, the end use of the software product, customized software, Technical Data and/or third-party software obtained from Intergraph Corporation, its subsidiaries or distributors is related to the design, development, production, or use of missiles, chemical, biological, or nuclear weapons, or other un-safeguarded or sensitive nuclear uses.

To any entity when Customer knows, or has reason to know, that an illegal reshipment will take place.

Any questions regarding export/re-export of relevant Intergraph Corporation software product, customized software, Technical Data and/or third-party software obtained from Intergraph Corporation, its subsidiaries or distributors, should be addressed to PPM's Export Compliance Department, 305 Intergraph Way, Madison, Alabama 35758 USA or at exportcompliance@intergraph.com. Customer shall hold harmless and indemnify PPM and Hexagon Group Company for any causes of action, claims, costs, expenses and/or damages resulting to PPM or Hexagon Group Company from a breach by Customer.

Trademarks

Table of Contents

Preface

This document is a user's guide for Smart P&ID 2019 automation programming course.

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

CHAPTER 1: 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 Smart 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 C# 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 C#.

◊ 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 SPID. 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 C# class.

◊ The Get function is typically defined for the Property, enabling the C# user 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 Smart 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 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.

3.4. *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.

3.4.1. 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.

3.4.2.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.

3.4.3.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.

3.4.4.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 Smart P&ID Plant, user must set a the exactly name of the Plant to the ProjectNumber of the LMADatasource

◊ There are two sources to get LMADatasource. First is by setting a new LMADatasource, second is get from SPID 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 SPID.

◊ New LMADatasource is a static query to the database, which is any changed made after the object is initialized, it won't know. In addition, if new LMADatasource modifies the items in active drawing, then user cannot modify the same item, otherwise, the drawing will crash due to inconsistency between database and drawing.

◊ Locale is another important property for LMADatasource, if set to "-1", it will make huge difference on performance at a satellite side, where the query will be conducted as Rep Schema locally instead of querying host site.

◊ 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.

◊ LMAAttributes is collection of LMAAttribute. LMAAttributes has a hidden property named “BuildAttributesOnDemand”, by default, it is set to False, but it’s recommended to set it to TRUE if only few attributes are accessed by the program, which is normally the case. However, by doing so, the attribute name becomes case sensitive.

> ☞ Note: Since V2007, the BuildAttributesOnDemand is by default set to TURE, hence the attribute name is always case sensitive.

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: Access LMAAttributes Collection.

Lab 10: Access ItemAttributions in Details.

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 11: Access an item using filters with single criterion.

Lab 12: Access items using filters with multiple criteria.

Lab 13: Access items using filters with criterion on Select List Data.

Lab 14: Using Compound Filter

Lab 15: Access all filters defined in the plant.

Lab 16: Access SelectList Data.

Lab 17: Create Filter with Select List Data in Criteria.

7. REVIEW

CHAPTER 2: SPID DATA MODEL

1. OBJECTIVES

In this chapter, you will learn

◊ The general structure of the Smart 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.

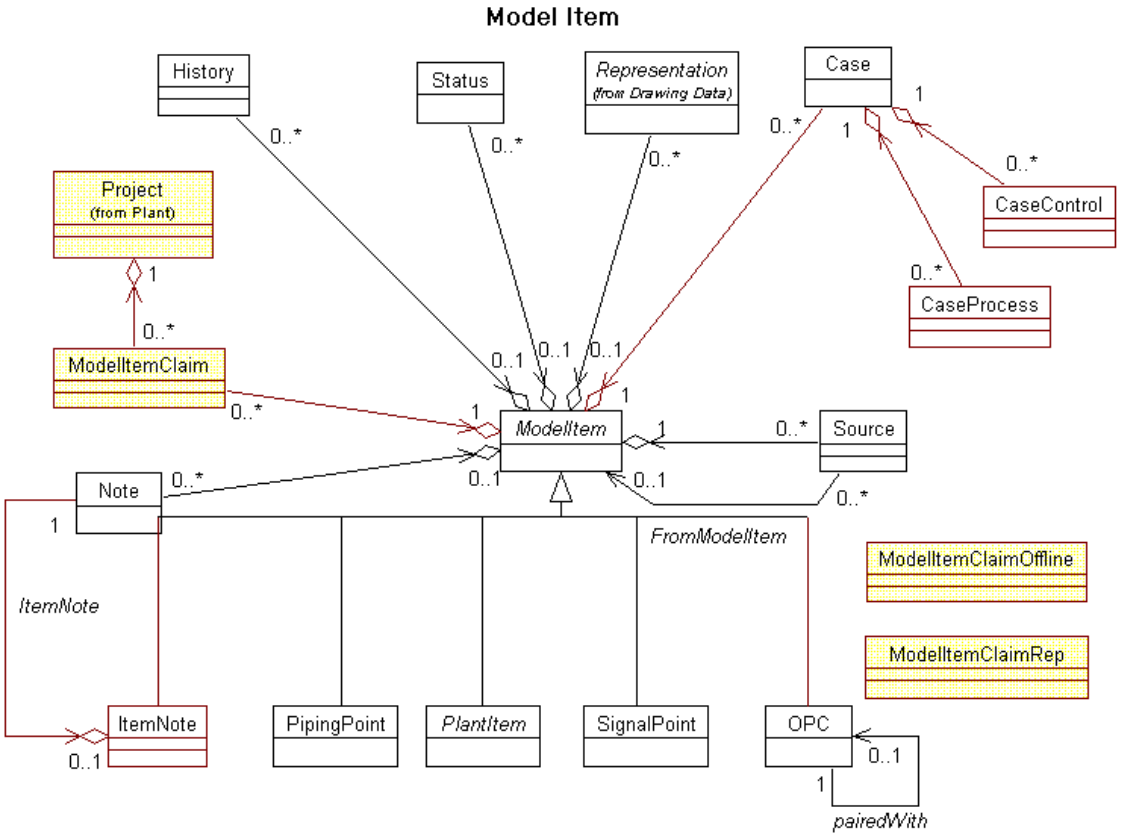
Lab 19: Read Status Property of ModelItem.

Lab 20: Read Case Property of ModelItem.

Lab 21: Access ItemNote.

Lab 22: Access OPC.

Lab 23: Filter for Histories



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.

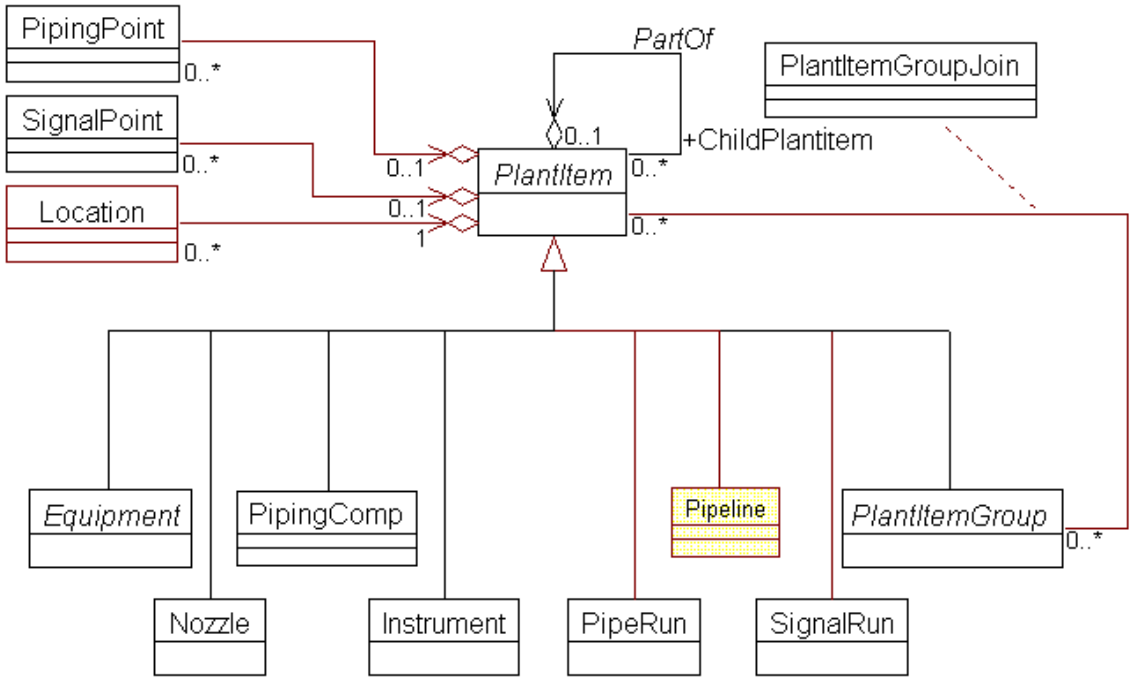
Lab 29: Implied Items.

Lab 30: Part of PlantItem relationships.

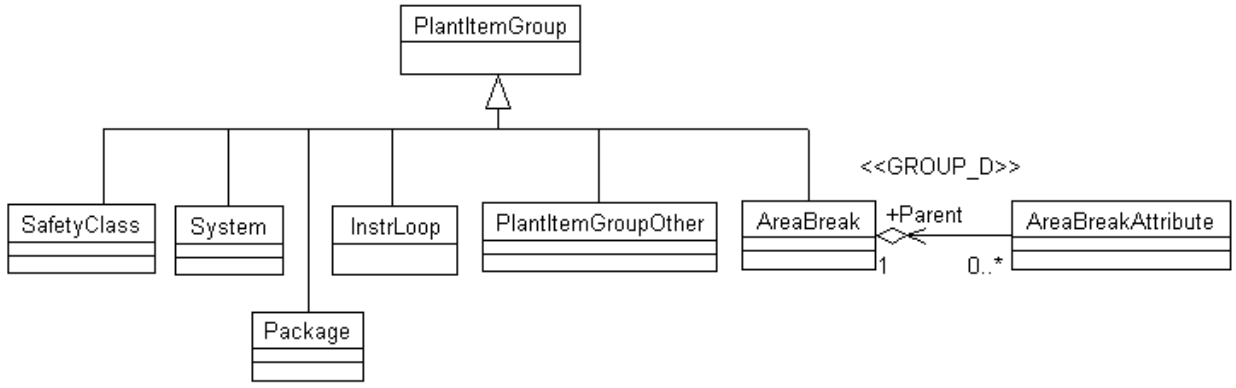
Lab 31: Access Instrument Loop.

Lab 32: LoadInstruments.

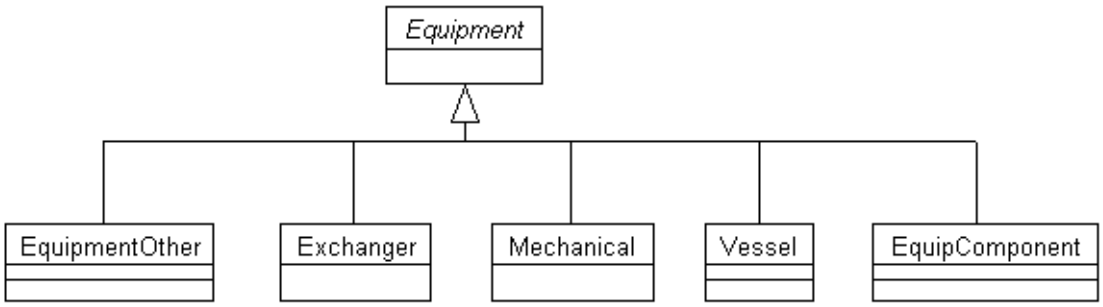
Plant Item Model



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

When a SignalRun is connected with an off-line instrument, SP_SignalRunID in T_Instrument table will be populated with SP_ID of the SignalRun.

3.4.6.Labs

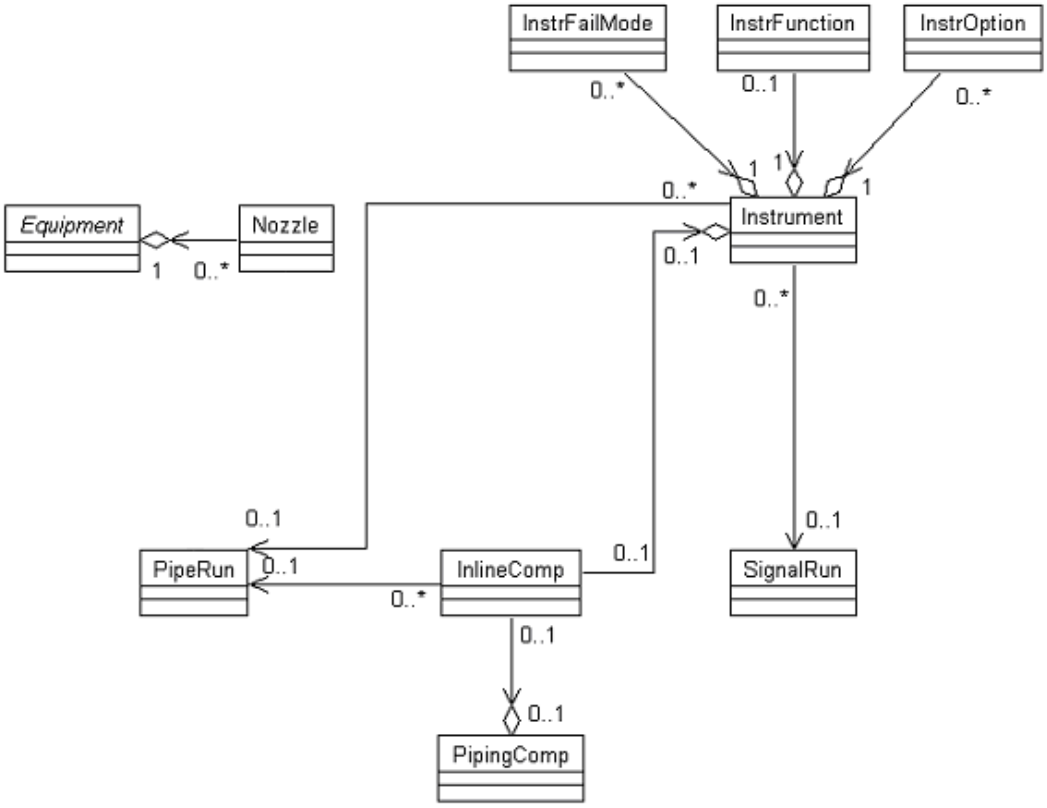
Lab 33: Identify Nozzle and Equipment.

Lab 34: PipingComp and InlineComp.

Lab 35: Instrument and InlineComp

Lab 36: Offline Instrument and SignalRun.

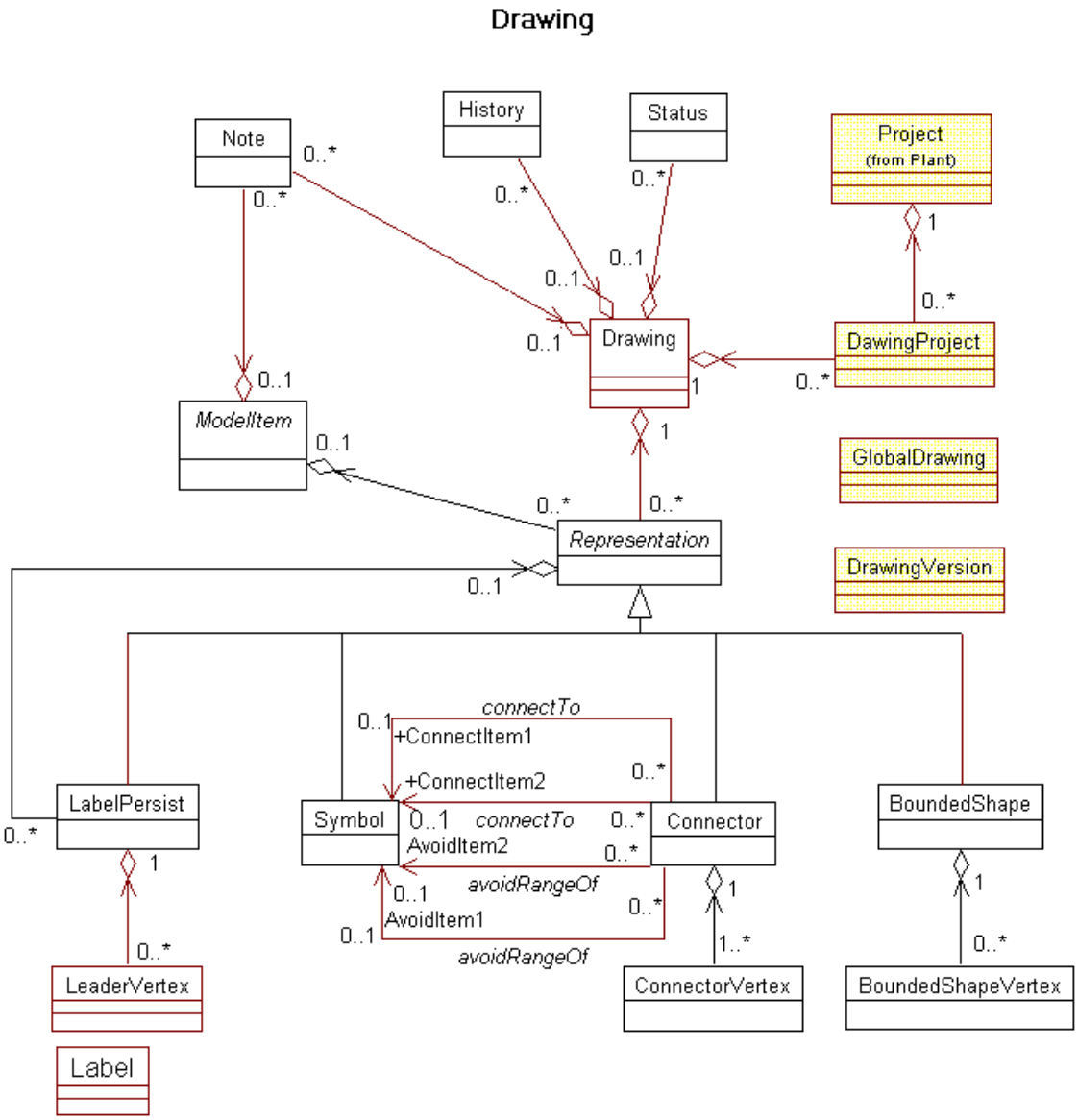
Lab 37: Access Instrument Functions.



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.

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.

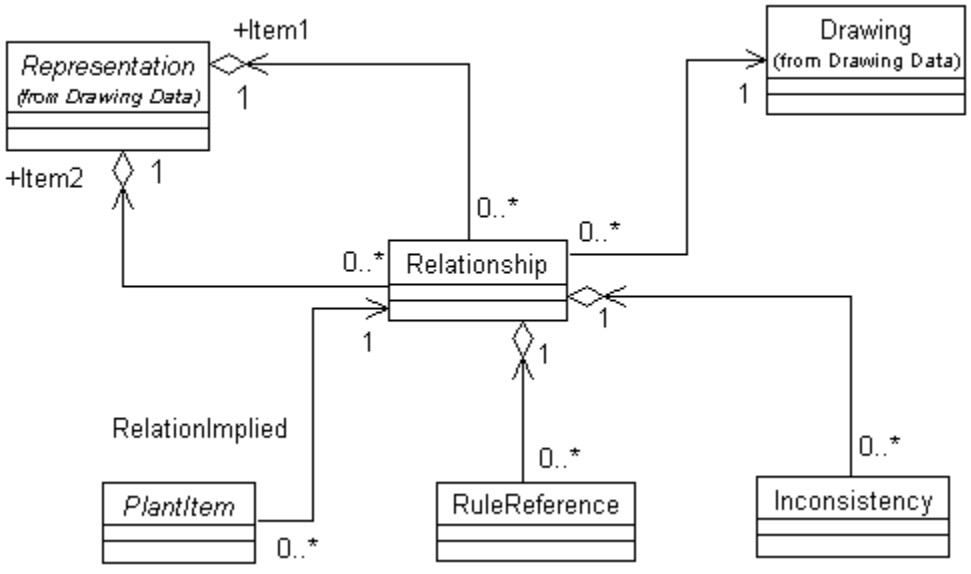
Lab 51: Navigate through OPC

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 SPID 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":

```
LMAItem 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

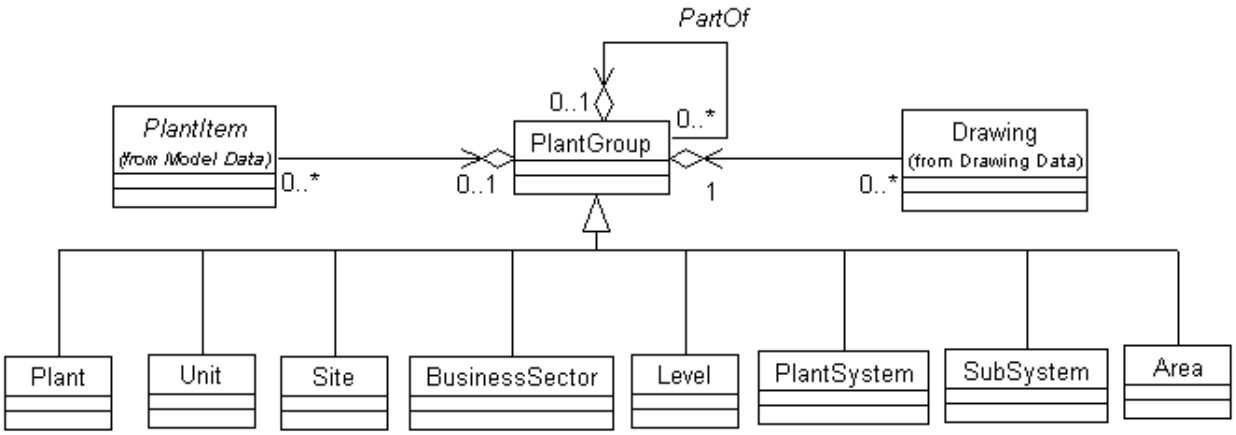
A collection of Drawings that are directly under the current PlantGroup.

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 SPID 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

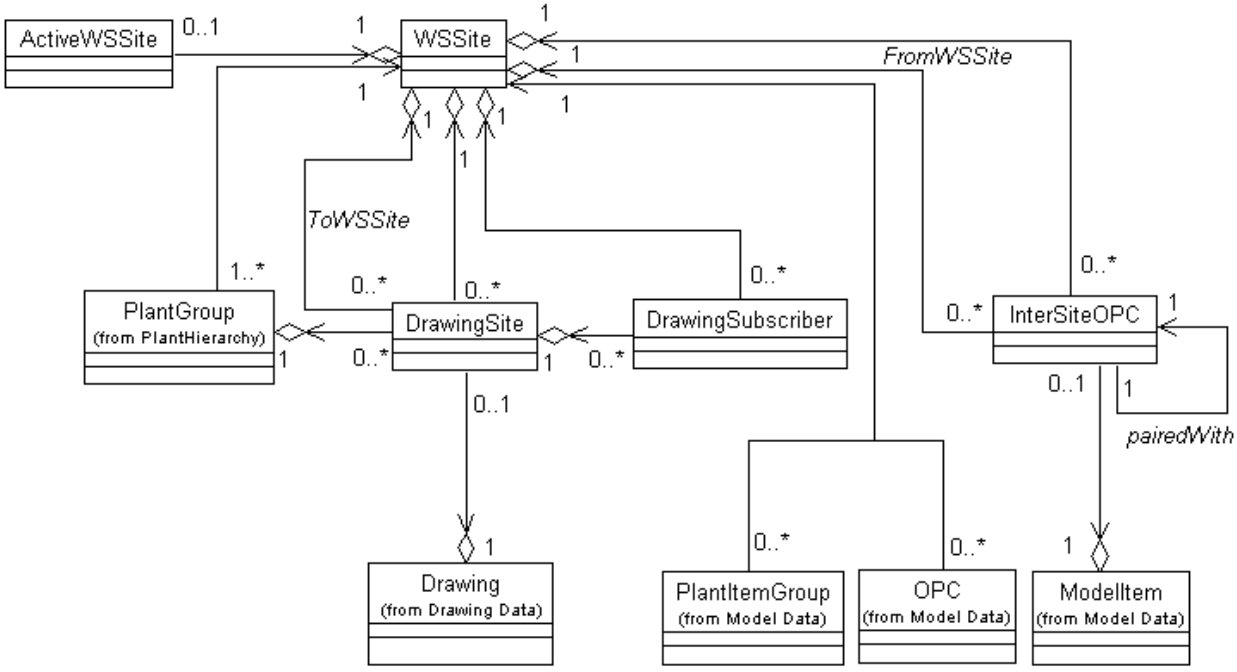
T_PlantGroup has a point that points back to the workshare site.

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

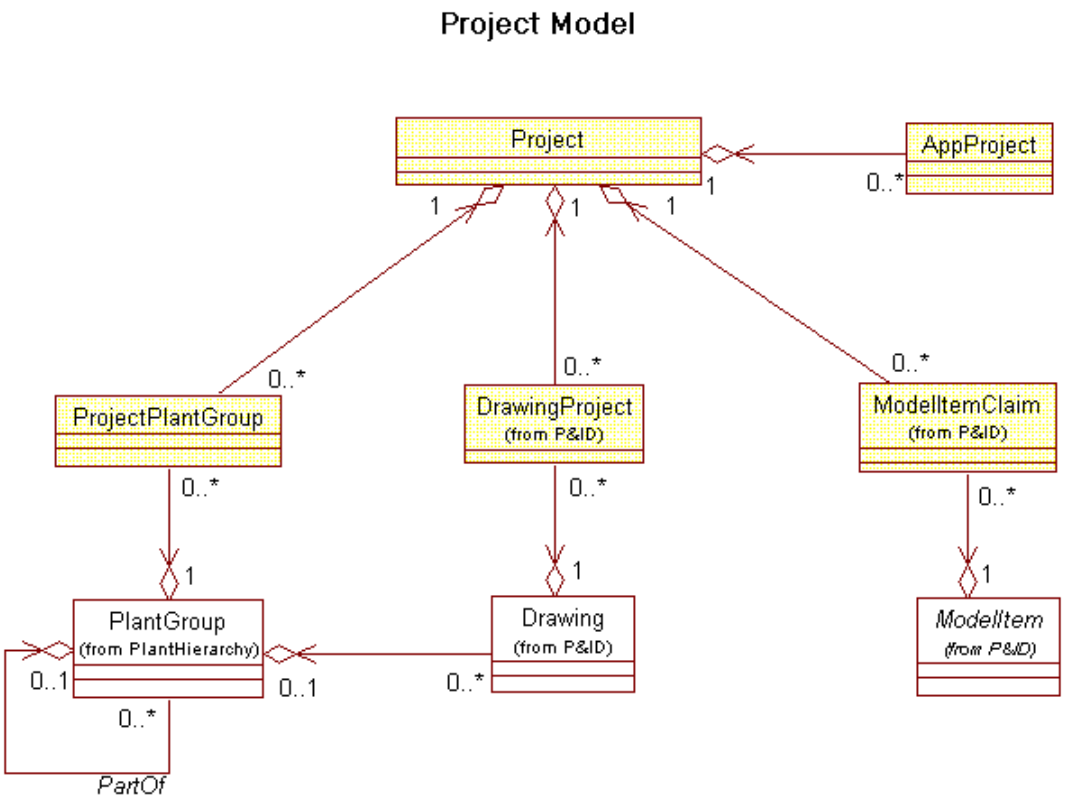
In As-Build/Project environment, an item can have different claim status: Not Claimed, Claimed by active project, and Claimed by Others.

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

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 3: PLACEMENT AUTOMATION

1. OBJECTIVES

In this chapter, you will learn

◊ An overview of the Placement Automation Plaice 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 Plant 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 Plant 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. *Join Piperuns*

Sub PIDJoinRuns(PipeRun1 As LMAItem, PipeRun2 As LMAItem)

PipeRun1: First Piperun as LMAItem to merge
PipeRun2: Second Piperun as LMAItem to merge

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

2.15.1. 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.

2.16. *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.

CHAPTER 4: USING PIDAUTO TO CREATE, OPEN AND CLOSE DRAWINGS

1. OBJECTIVES

In this chapter, you will learn

◊ Using PIDAuto Application to create, open and close drawings.

2. THREE IMPORTANT FUNCTIONS

2.1. *PIDAuto.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. *PIDAutO.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. *PIDAuto.Drawings.CloseAll*

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

CHAPTER 5: 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 Smart P&ID at specific events. This interface is supplied in the LMForeignCalc assembly. `Using ILMForeignCalc` statement incorporates this interface into the Active-X component. The functions are described below:

2.1. *DoCalculate*

◊ bool ILMForeignCalc.ILMForeignCalc.DoCalculate(LMADataSource DataSource, LMAItems Items, string PropertyName, ref object Value)

◊ 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.

◊ SPID 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*

◊ bool ILMForeignCalc.ILMForeignCalc.DoValidateItem (LMADataSource DataSource, LMAItems Items, ENUM_LMAValidateContext Context)

◊ 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.

◊ SPID 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

◊ SPID 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.

◊ In V2009 or later versions, The user can now create custom validation programs which will execute when certain drawing events occur: Open, Close, Create, Delete, Modify, and Print a drawing. These programs will reference ForeignCalc.dll and other automation dlls as needed. The user can filter on the following cases/events Open, Close, Create, Delete, Modify, and Print and add desired code for each case. A ProgID will need to be added for the **Drawing** ItemType in DataDictionary Manager to fire off the validation program.

2.3. *DoValidateProperty*

◊ bool ILMForeignCalc.ILMForeignCalc.DoValidateProperty(LMADataSource DataSource, LMAItems Items, string PropertyName, ref object Value)

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

◊ SPID 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 SPID accepts the user's entry into the memory.

2.4. *DoValidatePropertyNoUI*

◊ void ILMForeignCalc.ILMForeignCalc.DoValidatePropertyNoUI(LMADataSource DataSource, LMAItems Items, string PropertyName, ref object Value)

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

◊ SPID 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.

Lab 91: Create a Drawing Validation Program

CHAPTER 6: DELIVERED SOURCE CODE

1. OBJECTIVES

In this chapter, you will learn

◊ The delivered source code for some programs/utilities.

2. DELIVERED VALIDATION PROGRAMS

2.1. *PlantItem Validation*

2.1.1. This validation program gets fired up when an Item associated with ProgID in its validation program field is Create, Copy, Modify, Delete or Filtered.

2.1.2. If it is Create, it fires up ValidateUnit.Unit to the Item with PlantGroup-Unit.

2.1.3. If it is Copy, it fires up ValidateUnit.Unit to the Item with PlantGroup-Unit, and blank the value of TagSequenNo and LoopTagSuffix, then it fires up ItemTag.ItemTagFunc validation program.

2.1.4. If it is Filtered, it fires up ItemTag.ItemTagFunc validation program.

2.1.5. There is no validation as delivered if it is Modify or Delete.

2.2. *Unit Validation*

2.2.1. This validation program gets fired up when PlantItem validation programs call it internally, there is no UI for user to set it up to be fired up as other validation programs.

2.2.2. It checks if the Item being validated belongs to a drawing that is created under a PlantGroup, if Yes, it gets the SP_ID value and populates SP_PlantGroupID ItemAttribution for the Item being validated.

2.3. *ItemTag Validation*

2.3.1. This validation program can be fired up when a property is modified and/or calculation button is clicked if that property is associated with a ProgID in its validation field and/or calculation field. Then,

2.3.2. It generates new ItemTag or update existing ItemTag for Equipment, Piperun, Instrument, Signalrun, Instrument Loop and Nozzle.

2.3.3. This validation program has three different scopes regarding uniqueness check against items if it is project enabled. Scope 1 is when the uniqueness is against the active project only, it could be as-built, or one of the projects. Scope 2 is the active project plus as-built. Scope 3 is all projects plus as-built no matter what project the program is in.

2.3.4. Equipment ItemTag validation works as following way:

2.3.4.1. Changes on these properties will fire up validation program: TagPrefix, TagSequnceNo, and TagSuffix

2.3.4.2. It generates ItemTag in following format: (TagPrefix)-(TagSequnceNo)(TagSuffix)

2.3.4.3. It implements such mechanism that every Equipment gets a unique ItemTag, except statements in 5 & 6

2.3.4.4. TagSequnceNo will be generated or updated based on number specified in OptionSetting.
2.3.4.5. It validates all Equipment' ItemTag except the EquipmentComp
2.3.4.6. If it is a Equipment that has PartOfPlantItem relationship with other PlantItem (Parent), this Equipment could have a duplicated ItemTag with its parent.

2.3.5. Piperun ItemTag validation works as following way:

2.3.5.1. Changes on these properties will fire up validation program: TagSequnceNo, TagSuffix, OperFluidCode and UnitCode.
2.3.5.2. It generates ItemTag in following format:
(UnitCode)(TagSequnceNo)(TagSuffix)-(OperFluidCode)
2.3.5.3. Although no validation program is associated with UnitCode directly, ItemTag validation program gets fired up when UnitCode is modified, which is through Filtered action of PlantItemValidation.Validate validation program.
2.3.5.4. It checks if every Piperun gets a unique ItemTag, if not, user may select to keep duplicated or get a unique one
2.3.5.5. TagSequnceNo will be generated or updated based on number specified in OptionSetting.

2.3.6. Signalrun with PlantItem Type is Pipe Run

2.3.6.1. It is Piperun with Pipe Run Class is Instrument.
2.3.6.2. Works exactly same way as Piperun ItemTag validation

2.3.7. Signalrun ItemTag validation works as following way:

2.3.7.1. Signalrun ItemTag validation actually shares the same validation function with Instrument, but with fewer properties associated with Signalrun.
2.3.7.2. Changes on these properties will fire up validation program: TagSequnceNo and TagSuffix.

2.3.8. Instrument ItemTag validation works as following way:

2.3.8.1. There are two ways to fire up Instrument ItemTag validation as described in 2 & 3.
2.3.8.2. Changes on these properties will fire up validation program: MeasuredVariableCode, InstrTypeModifier, TagSequnceNo, LoopTagSuffix, and TagSuffix.
2.3.8.3. More likely, Instrument ItemTag validation program gets fired up when user pick up an Instrument Loop for the instrument. When user does so, by default rule, MeasuredVariableCode is copied from Instrument Loop's Loop Function, TagSequnceNo is copied from Instrument Loop' TagSequnceNo, and LoopTagSuffix is copied from Instrument Loop's TagSuffix
2.3.8.4. It generates ItemTag in following format:
(MeasuredVariableCode)(InstrTypeModifier)(TagSequnceNo)(LoopTagSuffix)(TagSuffix)
2.3.8.5. It checks if every Instrument gets a unique ItemTag, if not, user gets a warning, but the duplicated ItemTag will be persisted.
2.3.8.6. As described in 3, TagSequnceNo is copied from Instrument Loop to which the Instrument belongs.

2.3.9. Instrument Loop ItemTag validation works as following

2.3.9.1. Changes on these properties will fire up validation program: LoopFunction, TagSequnceNo and TagSuffix
2.3.9.2. It generates ItemTag in following format: (LoopFunction)-(TagSequnceNo)(TagSuffix)
2.3.9.3. It implements such mechanism that every Instrument Loop gets a unique ItemTag.

2.3.9.4. TagSequnceNo will be generated or updated based on number specified in OptionSetting.

2.3.10. Nozzle ItemTag validation works as following way:

2.3.10.1. Nozzle ItemTag validation validates uniqueness of Nozzle ItemTag per Equipment basis. In other words, all Nozzles that belong to an Equipment must have unique ItemTag, but Nozzles that belong to different Equipments can have duplicated ItemTag.

2.4. *OPC Validation*

2.4.1. This validation program is fired up when an OPC is Created, Copied, Modified, Deleted or Filtered. Then, it is performing ii.

2.4.2. Get a validate OPCTag for OPC and its paired OPC.

2.5. *Drawing Revision Validation*

This program validates the uniqueness of the drawing revision. If both major revision and minor revision are duplicated for a drawing revision, then this program will prevent the duplicated values accepted and give a message box to user indicating the problem.

2.6. *SyncSlopeRiseRunAngle*

This program is a calculation program for three properties of a PipeRun: Slope, Slope Rise and Slope Run. If any two values are given by user, this program will calculate the third value. In addition, if any given value is modified, this program will also re-calculate the other value accordingly.

3. IMPORT INTERFACE AND DELIVERED IMPORT CODE

This program imports foreign data into SPPID. The foreign data could be format of Excel reports, XML files, Text Files, etc.

Program has three major parts. First, read the data from external files, such as Excel, XML, or TXT files. Second, it searches the SPPID database and determine if there are matching items based on the criteria given in the program, such as SP_ID or ItemTag. Third, it will update the matching items or create a new item in stockpile if there is no matching item.

4. COPYTRANSOFMATION INTERFACE AND PROGRAM

This program provide an interface for user to control how the data in new drawing is created when copy/paste a drawing or when import a drawing. User can control what drawing template, data format, tag and plant group, etc. in new drawing.

5. LABS

CHAPTER 7: 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