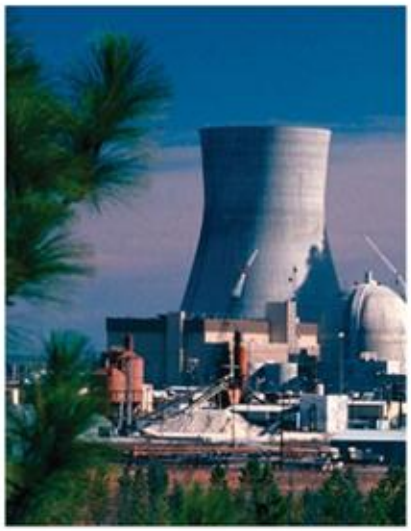
SmartPlant 3D

SP3D Automation using .NET – Course Guide

Process, Power & Marine


INTERGRAPH

Version 2009.1 **April 2010**

Copyright

U.S. Government Restricted Rights Legend

Intergraph Corporation
P.O. Box 240000
Huntsville, AL 35813
Street address: 170 Graphics Drive, Madison, AL 35758

Terms of Use

Warranties and Liabilities

Trademarks

S3D .net API

Overview of Smart 3D .net API

Topics

- S3D .net API Architecture.
- What Automation can you develop.
- API available for your use
  (to write Commands, Standalone Executables and Customization)
- Client Tier Services/Components
- Middle Tier Services/Components
- Commands – Modal, Non-Modal, Graphic, Step
- Metadata – Classes, BOCs, Interfaces, Properties, Codelists …
- Get/Set Properties, Access Relationships
- Writing Standalone Applications
- Application functionality
  - Common, Systems, Catalog, Eqp, Route, Space and an overall sample.

What is S3D .net API

- API → **A**pplication **P**rogramming **I**nterface

  is a set of *routines*, *data structures*, *object classes* and/or *protocols* provided by *libraries* and/or operating system *services* in order to support the building of applications.

- See http://en.wikipedia.org/wiki/API

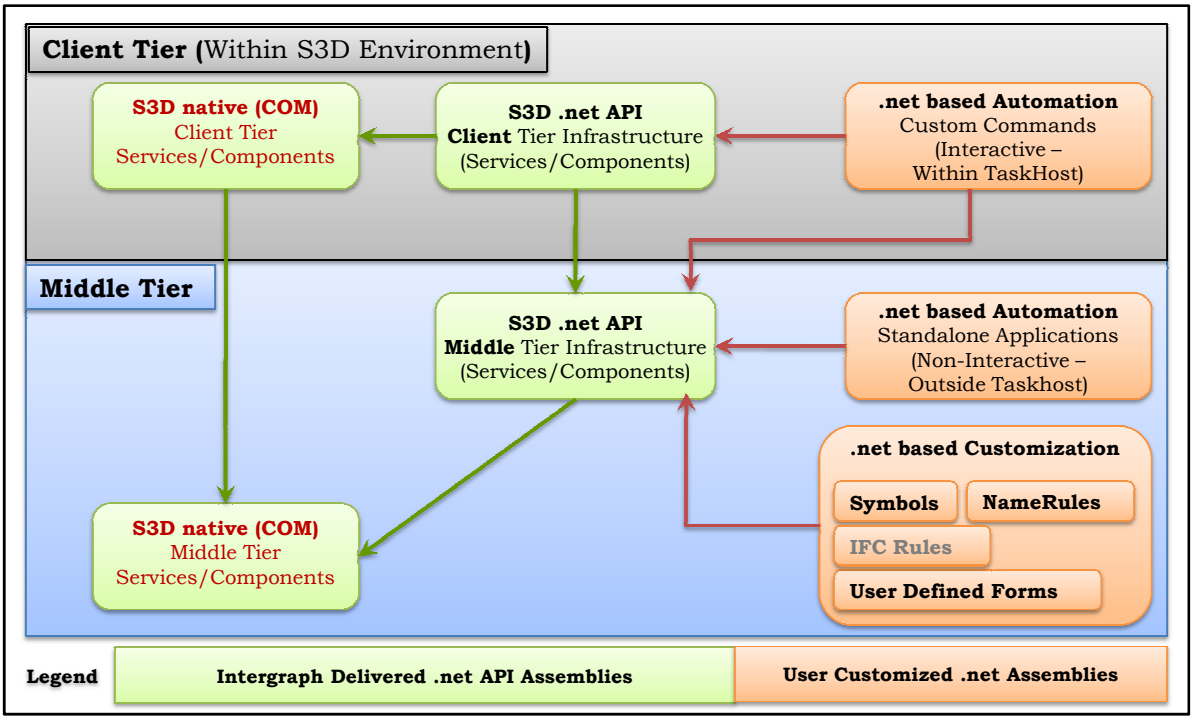
- S3D .net API → API of Smart 3D applications. (SmartPlant 3D & SmartMarine 3D)

S3D .net API Architecture

- Leverages **.net** methodologies, and provides **.net** based Automation capability.

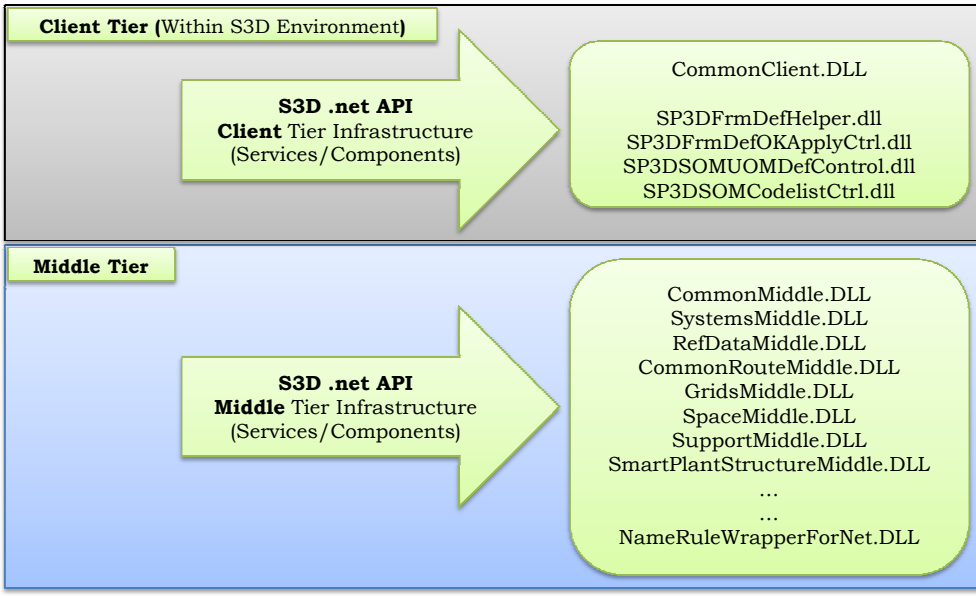
  Useful components delivered to you as **.net** Assemblies.
- You can program in **VB.net** or **C#** or any **.net** language.
- Use Visual Studio v2008.
- Delivered assemblies categorized as Middle, Client Tiers.
- Offers facilities to deal with both User Interface development for functionality within S3D; and Business Logic, to create/access/modify /delete objects and their data in S3D.
- Lets you write “interactive automation” – within taskhost, and, “non-interactive automation” – in custom executables outside of graphics environment.

S3D .net API Architecture



S3D .net API delivered Assemblies

All of the S3D .net API is delivered in assemblies in
$SmartPlant\3D\Core\Container\Bin\Assemblies\Release\



S3D .net API Highlights

- API available for use
  - Within TaskHost → Custom Commands
  - In customization → NameRule / Symbols / IFC Rule / …
  - In an external stand-alone application
- Fully documented with API Help integrated into Visual Studio Environment. F1 help works and shows context sensitive help.
- Examples provided for each aspect of the API.
- API for creating / modifying S3D objects, get/set properties, navigating relationships, access metadata.
- Utilize User Interface specific services – Graphics/Mouse/Keyboard/Selection/Visual Identification etc.
- Utilize Middle tier specific services/Functionality – Transaction control, Filters…

S3D .net API Highlights …

- Data integrity maintained with API usage to Create/Modify/Delete objects

Whether Smart3D .net API is used in TaskHost or a Standalone Executable,

the Semantic and Revision Management machinery is always

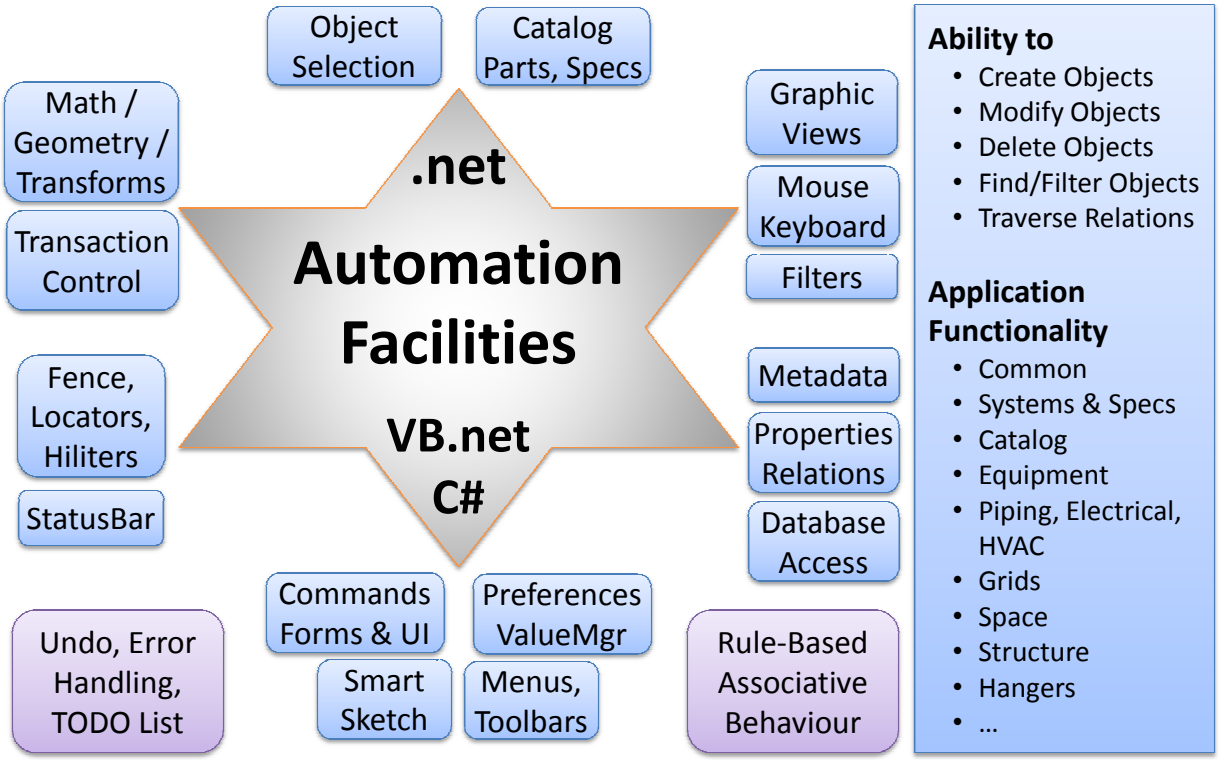
"running behind scenes",

percolating the changes to the required related objects,

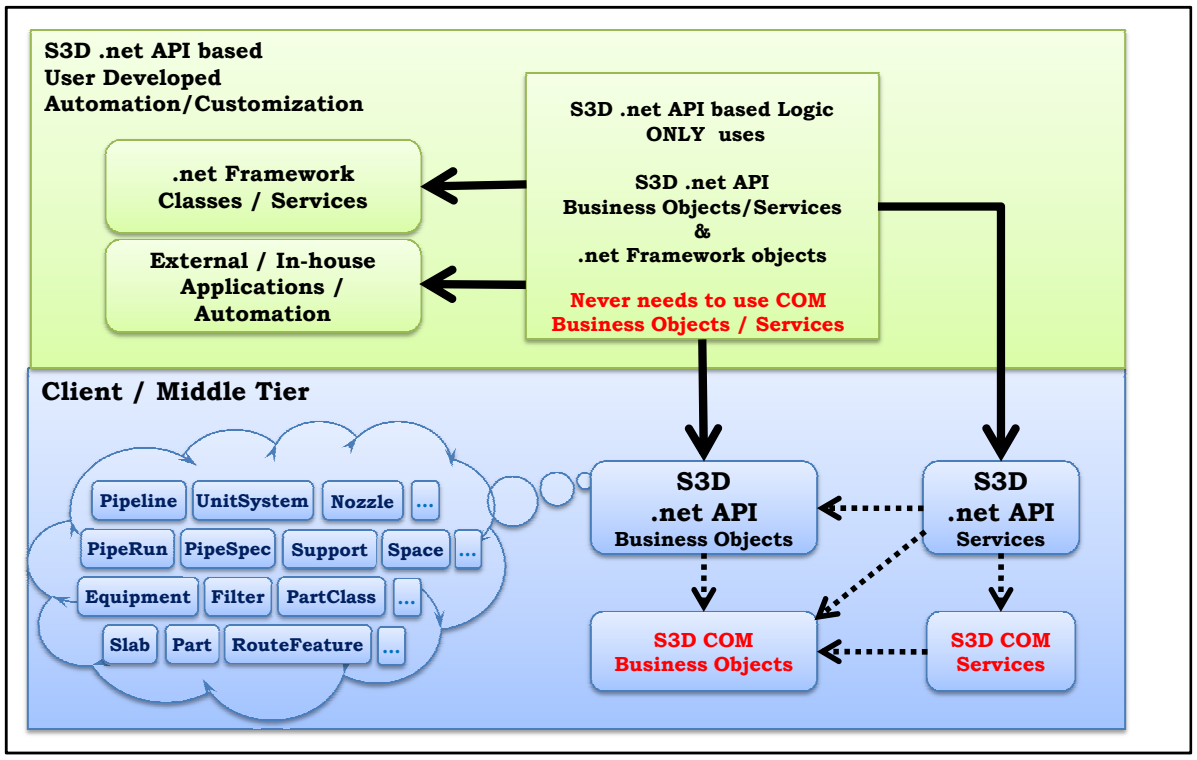
performing intuitive and implied updates to objects and related objects,

thus keeping the objects and data always in a *valid* state.

S3D .net API – Automation Facilities at a Glance



Business Objects / Services



S3D .net API

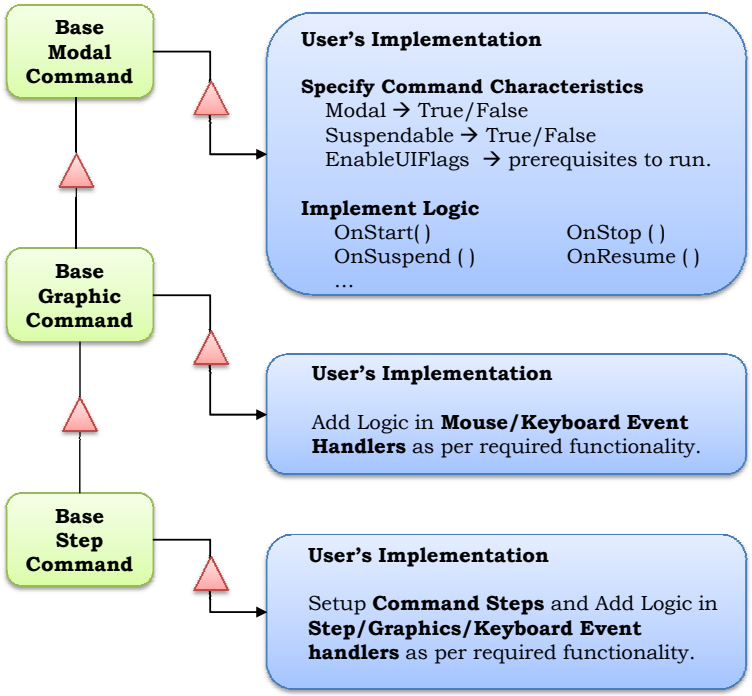
Commands

Types Of Commands

- **Basic Commands**
  - No further Graphic / Object inputs from Graphic Views/ Workspace Explorer after command is started.
  - Can be **Modal** or **Non-Modal**.
- **Graphic Commands**
  - Can use Graphic and Keyboard events.
  - non-modal, suspendable commands.
- **Step Commands**
  - Can use Graphic, KeyBoard and Object Selection events.
  - Allow gathering inputs (objects, points) in steps.
  - non-modal, suspendable commands.
- All these can get some inputs from a Form / RibbonBar.

User Implementation of Commands

- Based on the functionality, choose to inherit from one of
  - BaseModalCommand
  - BaseGraphicCommand
  - BaseStepCommand
- Implement required properties and methods.
- Override only required Methods & Properties, leave alone others.
- It is that simple !!!



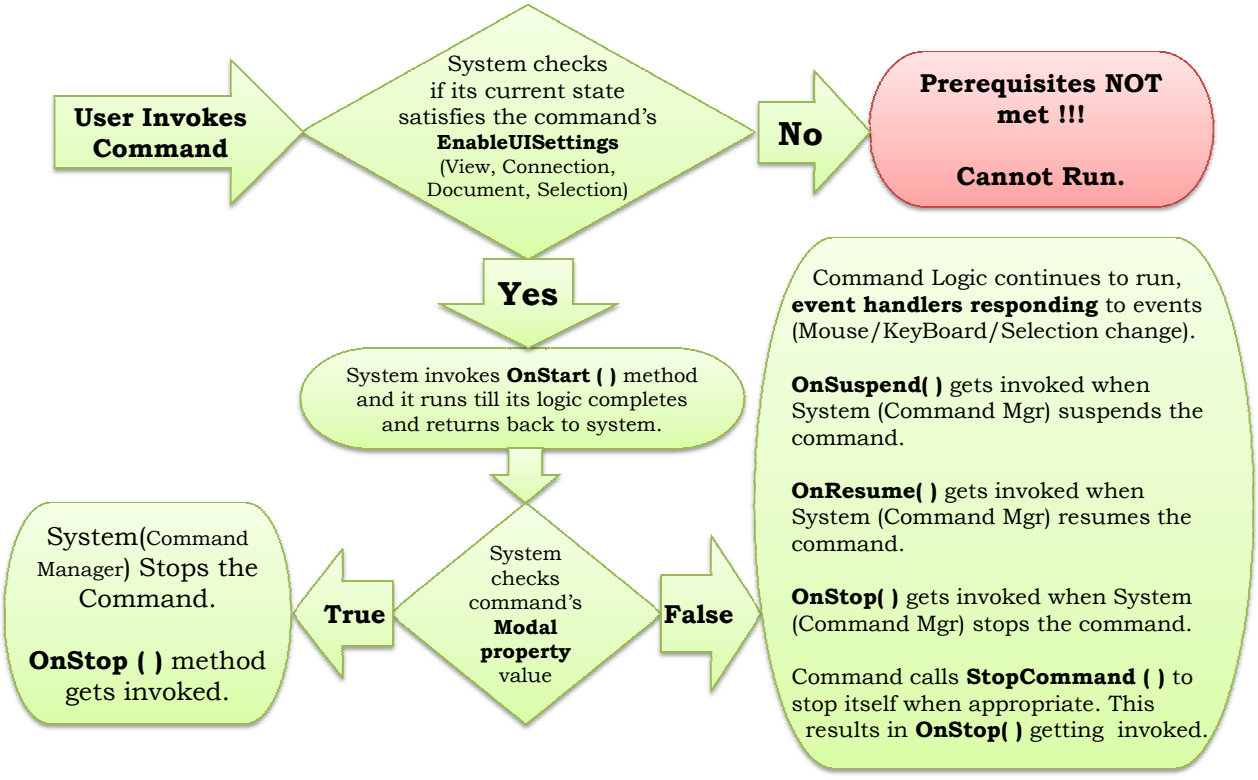
System & User's Handling of Command Types

- System calls **OnStart**( ) when it finally decides to start a command upon user's action. (more later on why system needs to decide whether to START or NOT)
- **Modal** commands → All functionality is within **OnStart**( ). When **OnStart( )** method returns, system Stops it (**OnStop( )** gets *invoked*).
- **Non-Modal** commands → Keep running until stopped (by itself or system).
- **Suspendable** commands → *Suspended* & *Resumed* based on the situation as system determines… (more details on this later in this session)
- Most Basic commands *typically* show up a form (modal / non-modal) to gather inputs like dimensions etc and do the processing inside the form's logic. – Form is not a necessity : eg Delete Command.
- **Graphic** Commands can get inputs (points from Graphic view) in addition to inputs from forms / RibbonBar / UI elements.
- **Step** Commands can get inputs (points/objects from Graphic Views, objects from Workspace Explorer), in addition to inputs from Forms / RibbonBar / UI elements.

Characteristics (Properties) of Command

- A command's **Modal** property specifies whether or not command expects more *graphic/object* inputs after invoked (can take some inputs via its form if any). If **True**, all command processing is within **OnStart( )**.
- A command's **Suspendable** property specifies if the command can be suspended temporarily while it is running, in response to a *higher priority* command being started. *Can* be specified as **True** if desired, only if its **Modal** property is False.
- A command's **EnableUIFlags** property specifies the pre-conditions to qualify before it run. Can choose one or more of below settings
  - No conditions needed.
  - Needs ActiveView → i.e. needs a Graphic View.
  - Needs ActiveConnection → i.e. defined workspace, connected to a plant.
  - Needs ActiveDocument → i.e. active session file.
  - Needs NonEmptySelectSet → i.e. some objects selected.
- Command Manager checks this setting with the system state to decide whether to start the command or not.

How System Runs a Command.



Command Implementation

- User just provides an override implementation for those properties/methods which he needs to do something. Doesn't need to implement all methods/properties.
  - **Modal** Property → Return True / False as appropriate
  - **Suspendable** Property → Return True / False as appropriate
  - **EnableUISettings** property → Specify prerequisite conditions required for the command to run. See EnableUIFlagSettings flags.
    - Specify any combination of ActiveView / ActiveConnection / ActiveDocument / NonEmptySelectSet flags. Default implies "None" i.e. "No Conditions required"

- Default Property values

Command → Property ↓	Modal	Graphic	Step
Modal	True	False	False
Suspendable	False	True	True
EnableUISettings	None	None	None

Command Implementation

  - **OnStart ( )** → Invoked by system when your command is started
    - for Modal Commands, this typically may do all the action, or shows a modal form which has all logic.
    - for Graphic / Step Commands, this typically does the initial setup and the event handlers have the functionality.
  - **OnStop ( )** → Invoked by system when it determines your command needs to be stopped.
    - Write your cleanup logic if any – unload forms / UI etc.
  - **OnSuspend ( )** → Invoked by *system* when *it* determines your command needs to be suspended …

    Here *typically*, the command must hide its UI form(s) etc and unsubscribe to events if any.
  - **OnResume ( )** → Invoked by *system* when *it* determines your command needs to be resumed …

    Here *typically*, the command must unhide its UI form(s) etc and re-subscribe to events if any.

Command Implementation …

- **OnIdle( )** → Invoked by *system* when *it* determines some idle time …

  Rarely commands have some implementation here, but you can do something like quick background processing of things without interfering with the main running functionality.
- To write to StatusBar, use the WriteStatusBarMsg (statusMsg) method. This is available on all Commands.
- For **Graphic** Commands, user implements required Graphic View Manager / Keyboard / RibbonBar event handling … (details in a later session)
- For **Step** Commands, user does Step Definition setup and implements Step / Graphic View Manager / Keyboard / RibbonBar event handling … (details in a later session)

Command Priority

- Commands are started with a preset priority, determined and requested by the starter (menuitem / toolbar icon / Custom commands dialog).
- Can be NormalPriority or HighPriority.
- HighPriority commands CANNOT commit/abort Transactions. Typically used for View manipulation, or any non-transacting non-dialog watch windows.
- Normal Priority used for most cases.
- Note : LowPriority has been deprecated and not to be used. May appear in the available options in the UI and the API enums, but still should not be used.

Command Suspension (stacking)

- Only one command is *active* at a time. (can have more command assistants running – more on Command Assistants later)
- Only one command could be stacked (i.e. put to suspended state) by System
- A command cannot suspend itself - ☺
- When a new command is invoked
  - If

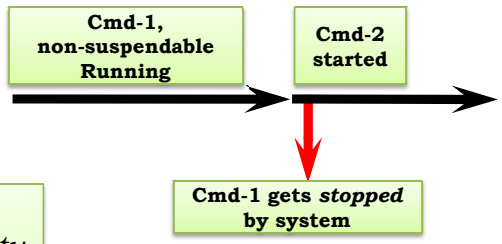
    CurrentCmd.Suspendable = True, and

    NewCmd.Priority > CurrentCmd.Priority
  - Then
    - Current Command is Suspended (**OnSuspend( )** is invoked), and
    - later resumed (**OnResume( )** is invoked) when New Command stops.
- All other cases, Current Command is stopped (**OnStop( )** is invoked) when a new Command is started.

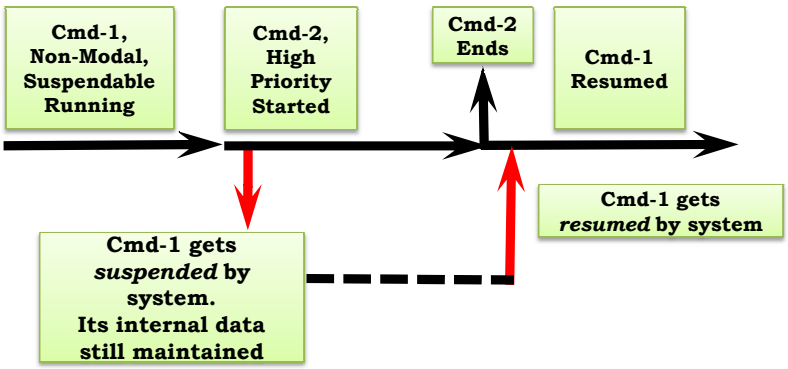
For this reason, making a High Priority command suspendable doesn't make any sense.

Command Suspension (stacking) …

A running ***non-suspendable*** command gets *stopped* by system when any another *command* is invoked.

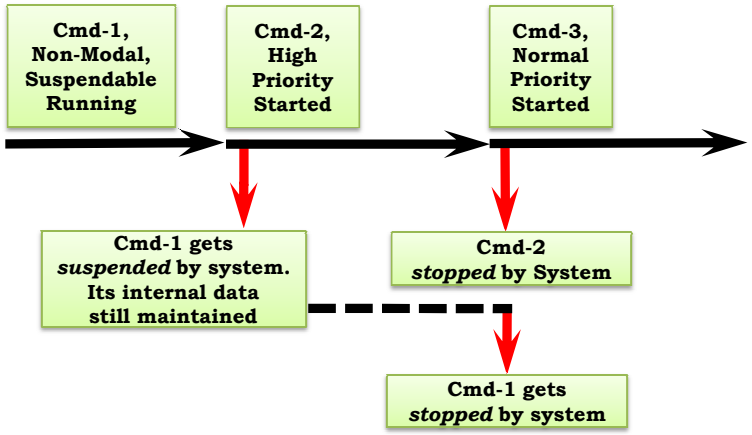


A running ***suspendable*** command gets *suspended* by system when any *High priority command* is invoked. System *resumes* the suspended command when the high priority command stops.



Command Suspension (stacking) …

A ***suspended*** command gets *stopped* by system when any *Normal priority command* is invoked.



Implementation Expectations

- Commands should not *generally* leave their modifications on the table, i.e. just modify things and leave them there for next command to commit to the database. i.e. either abort or commit the transaction (more on transactions later) before they are completely ended.
- Command should not *generally* inherit uncommitted changes done by other commands. Most cases, this can be simply handled by performing a TransactionMgr Abort in the initialization of the command. If it is a Helper command or a non-transaction based command, eg view manipulation etc, then it should not do an abort.
- There are few exceptions to above. In case of commands which do not perform any changes to business objects, or commands which do the changes and intentionally delegate the commit / abort responsibility to other commands (which they can start)
- Commands must withdraw their UI when they are suspended / stopped. There may be very rare exceptions to this. For example, the View Todo list command shows its form and the Form can be active after the Todo list command has exited. One can use other commands while the TODO list form is up.
- Wherever possible, all errors happening within the command logic must be handled internally in the command logic itself and not be propagated outward.

What is running at any time in the System ???

- At any time, “a Command” is always running in the system. There is never a state when there is no “running command” in the system.
- Yes, it is True !!! Such command is called the Default command, defined by the active Task (the Environment).
- It always runs when no command is running. Of course, it cannot be a Modal command. Most of the cases it is the “**Select Command**”.
- When Select Command senses something is selected, it starts off a “Modify Command” which runs in parallel with it –
  - A modify command is a special type of command (internally started by system).
  - An application’s logic at runtime tells the system what is the Modify Command for the selection state.
  - This usually posts a special ribbon bar which allows to modify currently selected object(s)
- When a command ends itself and there is no command in suspended state, the system starts the Default Command.
  - There exists an internal mechanism – called Command Continuation – which a Task can use to change this process as needed so as to restart the previously running command – this mechanism is not yet available for End User’s Automation as it is quite complex to use by end users.
  - Eg : Route Pipe → Insert Component & Finish → notice the Route Pipe command *continue*
- It is important to know this details because this knowledge helps to know the system’s behavior and write better automation.

S3D .net API

LAB

–

Writing Commands

LAB - Overview

- In this LAB we will learn
  - Writing Custom commands in VB.net
    - Creating Project, manage its settings
    - Adding required references,
    - Importing required namespaces,
    - Inheriting from one of the base command classes,
    - Providing override implementation – command functionality
  - Deploying Custom Commands
  - Running Custom Commands
  - Debugging Custom Commands
  - Study Command Characteristics
    - Modal/Non-Modal/Non-Modal & Suspendable commands

Create a VB.net Class Library Project

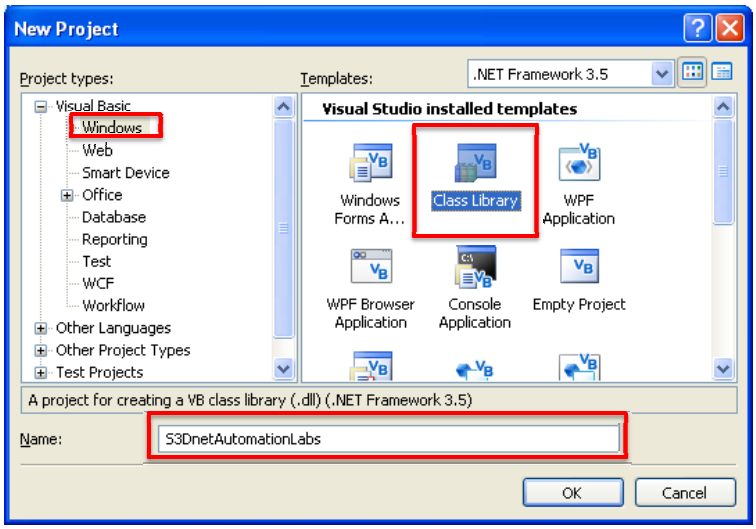
- Use Microsoft Visual Studio 2008.
  - Choose a .net language → VB.net / C#
    - VB.net is preferred. Most samples are in VB.net. Our examples will be VB.net.

- Project Type
  - Use a Windows "Class Library" project template
  - Can create the new command in
    - an existing **Class Library** project you already have with other commands you have developed.
      - Choose File → Open Project
    - a new **Class Library** Project.
      - Choose File → New Project

  As this is our first Lab Command we choose New Project.

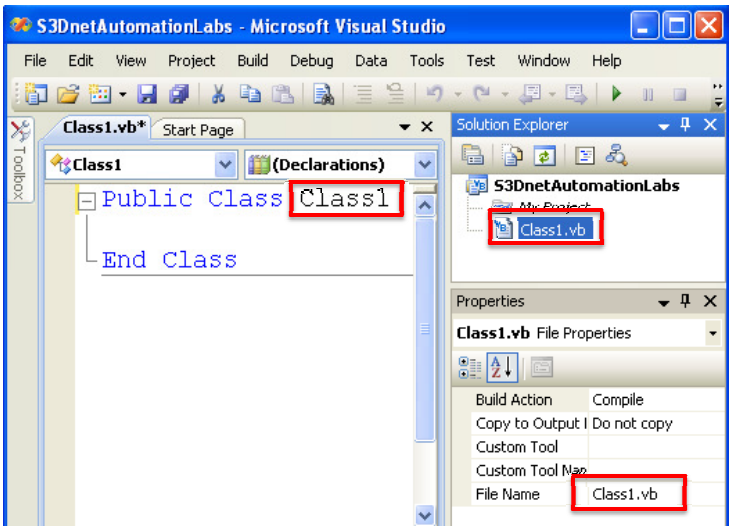
Create a VB.net Class Library Project

- Choose **File** > **New Project**
- Use **Visual Basic** > **Windows** from Project Types
- Pick **Class Library** template
- Specify **Name** of Project
- Hit **OK**.
- Project gets created.
- …

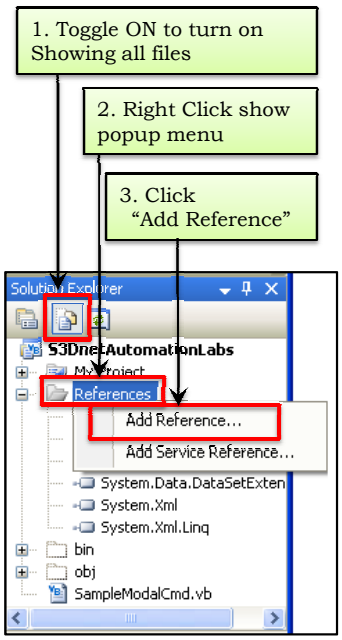


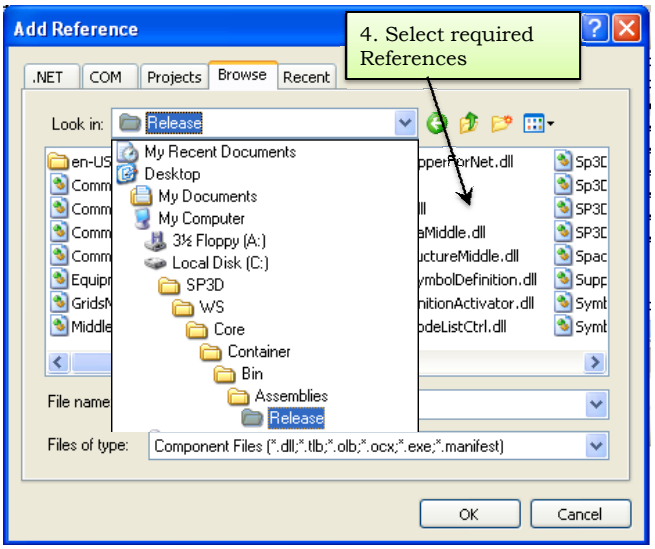
Create a VB.net Class Library Project

- It creates a Class1.vb which has an empty class named Class1.
- Rename the Name of the class from Class1 to a name of your choice and also rename the filename accordingly.
- Lets say we change it to "SampleModalCmd", for the sample **Modal** command we will be writing here.



Attach Required References

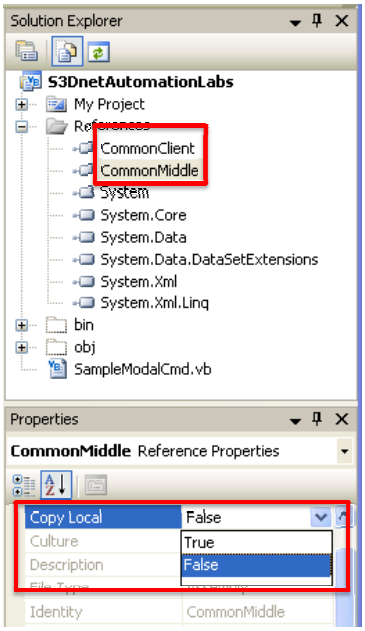




Select **CommonClient.DLL** and **CommonMiddle.DLL**, which are required for writing a basic command. You can choose other assemblies as needed for the functionality of the command.

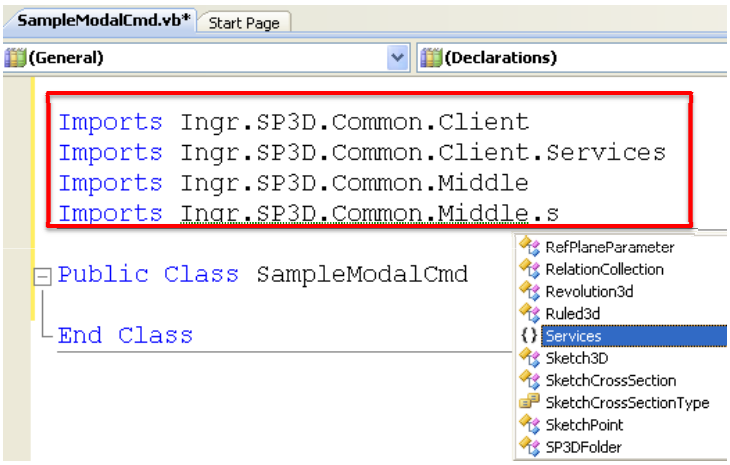
References : Copy Local setting

- Typically, Visual Studio adds references with "**Copy Local : True**" setting, i.e. it is copied locally for the project's use.
- Change it as "**Copy Local : False**" in the Properties tab of the assembly reference.
- Otherwise, when you install updates of the S3D product(s), your projects may continue using old copies of the references copied at the time of adding the reference.



Import Required Namespaces

- Add the required "Imports" statements at the top of the file to enable you use the types in that namespace without using the full name of the class when needed.



- This lets you just use

Dim oSelectSet as **SelectSet**

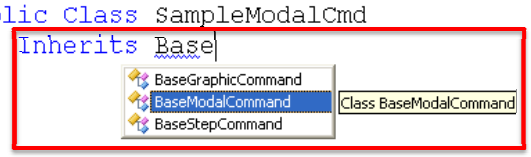
instead of

Dim oSelectSet as **Ingr.SP3D.Common.Client.Services.SelectSet**

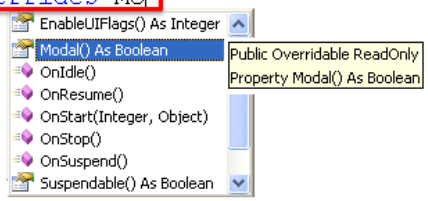
Inherit from a Base command class

- To be runnable inside S3D, Commands must ***inherit*** from one of the command base class types
  - Modal / Graphic / Step
- We then provide the ***override implementation*** for the properties/methods, i.e. the command characteristics and functionality.
  - **Modal, Suspendable, EnableUIFlags**
  - **OnStart( ), OnStop ( )**
  - **OnSuspend( ), OnResume( ) (for suspendable commands)**

```
Public Class SampleModalCmd
    Inherits Base
End Class
```




```
Public Class SampleModalCmd
    Inherits BaseModalCommand
    Overrides Mo
End Class
```



Override Required Implementation

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)

    Dim oSelectSet As SelectSet = ClientServiceProvider.SelectSet
    If (oSelectSet.SelectedObjects.Count = 0) Then
        MsgBox("No Objects Selected")
    Else
        Dim strSelectedObjectNames As String = ""
        For Each oObj As BusinessObject In oSelectSet.SelectedObjects
            strSelectedObjectNames += vbCrLf & oObj.ToString
        Next
        MsgBox(strSelectedObjectNames, , "Selected Object Names")
    End If
End Sub
```

Guess what, we are done !!! ??? YES

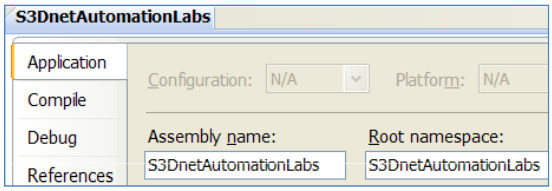
Lets see next page why we don't have to do any more overrides.

Override Required Implementation

- We don't have any specific **OnStop( )** implementation as there is nothing specific we want to do at the moment when the command is stopped.
  - Typically, one would want to implement cleanup code in the **OnStop( )**.
  - The BaseModalCommand class implementation of **OnStop( )** is good enough for our case and we don't need to provide an **OnStop( )**
- Also, since ours is a ***Modal*** command, there is no need to implement the ***Suspendable*** Property as well as the ***OnSuspend( )*** and ***OnResume( )*** methods, because the system never needs them for ***Modal*** commands. They can be implemented for ***Suspendable*** commands.
- Observe that we only "override" the required implementation, unlike in COM, we had to provide entire implementation. Thus, a big reduction in number of lines of code compared to COM.
- At this time, our command is ready to run or debug.

Deploying & Running Commands

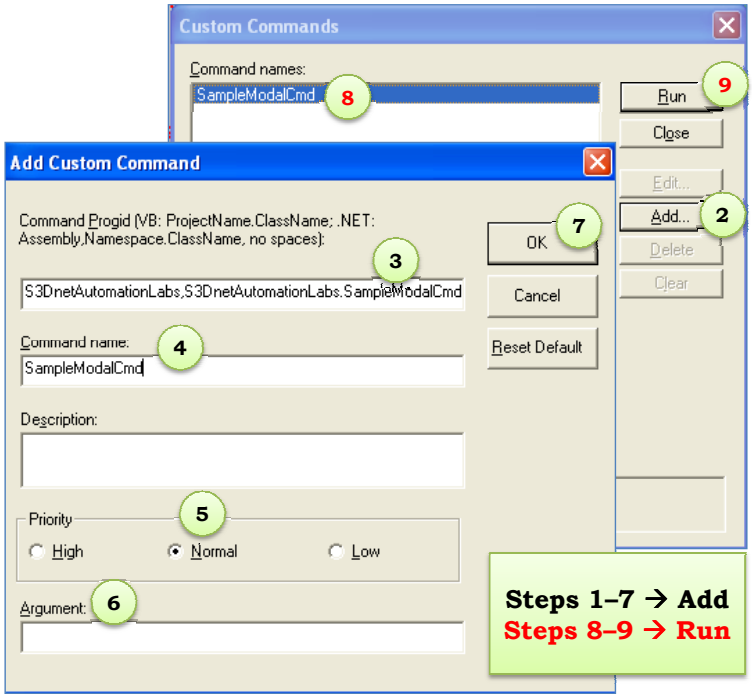
- Save the Custom Command Assembly DLL to a location on the disk.
- Invoke menu item **Tools > Custom Commands**
- Add command to list of custom commands if not already done. (more details discussed next). You will need
  - the **Assembly (DLL) Name** with full path but without extension,
  - the **NameSpace** of Command Class – if you have not explicitly created your class in a different namespace, it goes into project's Root NameSpace (see it in Application Tab of Project Properties).
  - the **ClassName**
  - Like *{FullPath}AssemblyName,NameSpace.Class*



- Select the command and then click **Run**. (details later in this session)
- Note : FullPath is optional if you are deploying into SymbolShare\CustomCommands, or a directory included in Environment Variable named S3DCustomAssemblyPaths. Can specify ***relative*** path with respective to those standard lookup directories.
- Other means to invoke commands exist – modifying XML files which control UI of the app. (eg CommonApp\Environment\Xml\Include\MenuTools.xml)
- <Menuitem caption="MyCmd" action="StartAssemblyCmdNormal" arg1="*{FullPath}AssemblyName,NameSpace.Class* "/>

Add/Run Command in SP3D/ SM3D

1. Within SP3D / SM3D, choose **Tools > Custom Commands**
2. Click **Add** to add your new command.
3. Provide **Command ProgID** as specified. (`AssemblyName,NameSpace.ClassName`)
   For our example the ProgID will be **S3DnetAutomationLabs, S3DnetAutomationLabs.SampleModalCmd**
   Note: Prefix full Path of the DLL folder for progID.
4. Provide **Command name**, as **SampleModalCmd** in our example.
5. Choose appropriate command **Priority** - Normal / High. We choose **Normal** for our example.
6. Provide Command **Argument** as required. We leave it blank for our example.
7. Click **OK** to add it. It then appears on the Custom Commands dialog.
8. To Run it, select it from Custom Commands dialog and press **Run**. 9



Moving/Copying/Opening Projects across machines

- **.NET** is not like **COM** in many respects.
- When you open a Project in a new machine, if the 3D installation folder in the new machine is not the same as that of the machine where the project was originally saved, you will encounter broken references.
- This is due to the fact that the references added to the project from the original machine are not found at those directory locations in the new machine.
- Whereas, in COM, the system goes by the registry to auto correct the DLL file references.
- To solve this situation, You will have to open and fix the project files in the new machine to use the corrected paths.
- Using Standard installation paths for the developers in your automation team may be a good idea to avoid this problem.
- Deployment is not an issue for custom commands. For Standalone EXEs, if you follow certain guidelines (explained later), then you do not have any problem at runtime.

Debugging the Command

Approach #1: Attach to an already running SP3D / SM3D process.

Approach #2: Specify Executable to start which invokes the command class. (details next)

Note : Developer Studio **Express Edition** limits its UI from doing both of these approaches. However, Approach #2 can be setup by specifying the details of the Executable in the <project>.vbproj.user file. Optionally you can also specify the Session File name in StartArguments section.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<PropertyGroup>
  <StartAction>Program</StartAction>
  <StartProgram>C:\SP3D\WS\Core\Container\Bin\TaskHost.exe</StartProgram>
  <StartArguments>FullPathToSessionFile</StartArguments>
</PropertyGroup>
</Project>
```

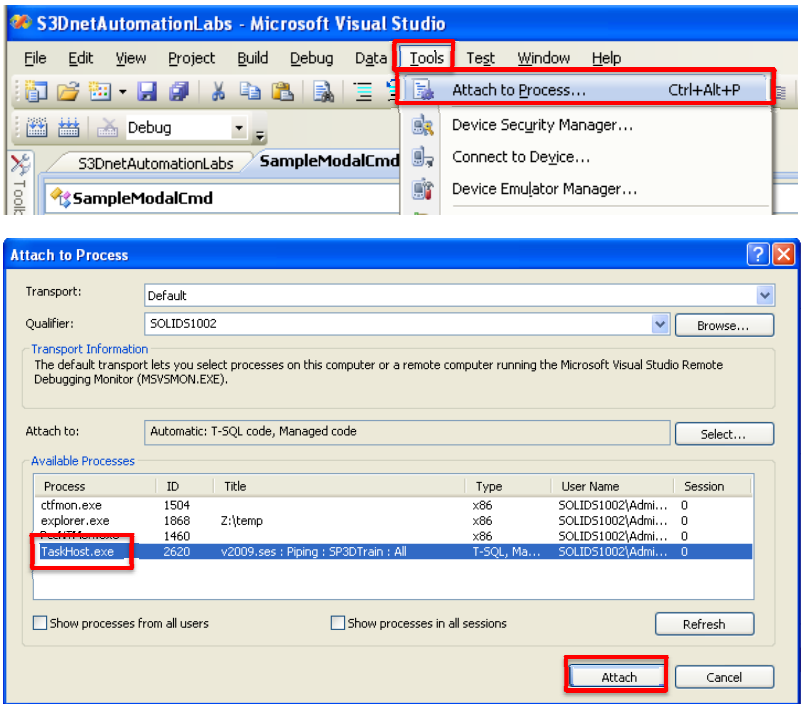

Debugging the Command

Approach #1: Attach to an already running SP3D / SM3D process.

Inside Visual Studio 2008, choose **Tools > Attach To Process**,

Select the T**askHost.exe** process and press **Attach**.

Inside TaskHost, invoke your command from **Tools > Custom Commands** as described earlier.



Debugging the Command…

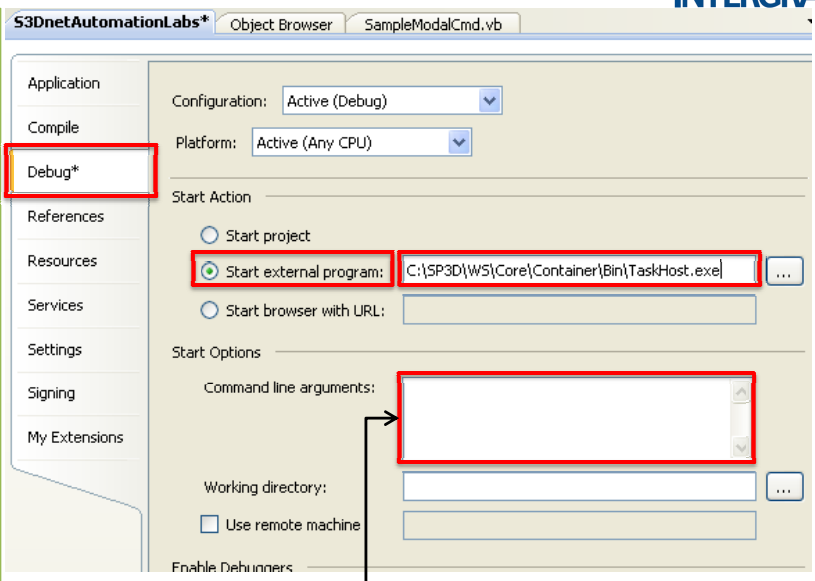
Approach #2: Start a new SP3D / SM3D process from within the debugger. (Note: you need to add the full **Core\Runtime** and **GeometryTopology\Runtime** paths to your Path Environment Variable first.

Inside Visual Studio 2008, **Project > Project Properties**

Choose the following settings and **save**.

Now select **Debug>Run menu item in Visual Studio.**

Invoke your command from **Tools > Custom Commands** as described earlier.



TIP : You can also specify the session file name in the **Command line arguments** field, with which the application will start in that session file directly instead of the normal open session file dialog.

Further improvements to Command

- Next, we enhance the command to specify some **EnableUIFlags**. We want our command to be enabled to run only when there is an **ActiveConnection** and **NonEmptySelectSet**. We do this by providing Override implementation of the ***EnableUIFlags*** property

```
Public Overrides ReadOnly Property EnableUIFlags() As Integer
    Get
        Return EnableUIFlagSettings.ActiveConnection + _
               EnableUIFlagSettings.NonEmptySelectSet
    End Get
End Property
```

- With this changes, system doesn't start our command when the command's EnableUIFlags property (i.e. its prerequisite expectations) does not conform to the system state at the time of invocation of the command.

Final code – SampleModalCmd.

```
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle
Imports Ingr.SP3D.Common.Middle.Services
```

1

```
Public Class SampleModalCmd
  Inherits BaseModalCommand
```

2

```
Public Overrides ReadOnly Property EnableUIFlags() As
      Integer
    Get
      Return EnableUIFlagSettings.ActiveConnection + _
          EnableUIFlagSettings.NonEmptySelectSet
    End Get
  End Property
```

4

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)
  MyBase.OnStart(commandID, argument)

  Dim oSelectSet As SelectSet = ClientServiceProvider.SelectSet
  If (oSelectSet.SelectedObjects.Count = 0) Then
    MsgBox("No Objects Selected")
  Else
    Dim strSelectedObjectNames As String = ""
    For Each oObj As BusinessObject In oSelectSet.SelectedObjects
      strSelectedObjectNames += vbCrLf & oObj.ToString
    Next
    MsgBox(strSelectedObjectNames, , "Selected Object Names")
  End If
 End Sub
End Class
```

3

Inheriting from **BaseModalCommand** defaults the **Modal** property to **True**. Overriding **Modal** Property is only needed if you want to create a Basic Non-modal Command.

```
Public Overrides ReadOnly Property Modal() As Boolean
    Get
        Return  False
    End Get
End Property
```

We extend this lab little to study how non-Modal, Suspendable commands work.

Further study : Command Characteristics

Next, we study a little about Modal & Suspendable command behavior.

- If we made our Command Non-Modal, then what happens after OnStart() method completes i.e. after the message box is shown ???
  - Which command is running at this moment ?
  - Which command was running at this moment, when our command was Modal ?
  - what happens if we invoke and end the view zoom command ? Which command is running now ?
- Additionally, what new behavior do you see if we make our command Suspendable as well ???
  - With our command made non-Modal and Suspendable – what happens if we invoke and end the view zoom command ? Which command is running now ?
- It would be little more clear if we override the OnSuspend, OnResume and OnStop methods to indicate in status bar message and / or message box to indicate they have got invoked …
- This is left as an exercise to the class students …

S3D .net API

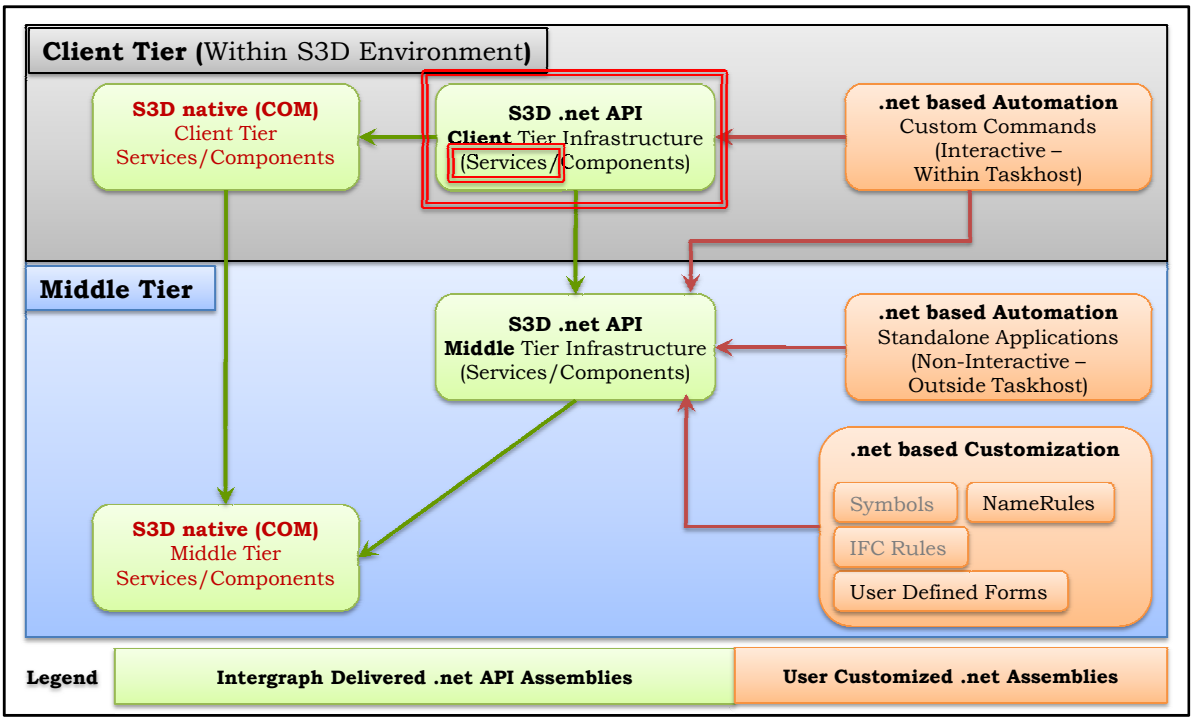
Client Services

-

Automation Services

Related to User Interface

S3D .net API – Client Services



What is a Service

- A service is something which provides a useful functionality. Often a service is a unique running component, providing *programmatic* access to needed functionality. For example
  - SelectSet service provides selected objects related functionality.
  - TransactionManager service provides transaction related functionality, i.e. commit, compute, abort a set of changes to business objects.
  - GraphicViewManager service provides functionality related to Graphic views, Events, etc.
  - …

CommonClient Services

- *ClientServiceProvider*, a *static* class, provides access to needed services, which in-turn provide access to components and other artifacts useful in Client Tier
  - SelectSet
  - TransactionManager
  - GraphicViewManager
  - WorkingSet
  - ValueMgr
  - Preferences
  - GraphicView
  - Hiliters
  - Locators
  - RectangularFence
  - SmartSketchOptions
  - Commands
  - BOCollections

Available in **CommonClient.DLL**, namespace : **Ingr.Common.Client.Services.**
Being a static class, **ClientServiceProvider** can be used directly without the need to **new** it.
Never use **New** operator to get a Client service. Always get it from **ClientServiceProvider**.

SelectSet

- SelectSet service provides programmatic access to element selection.
- Accessible as **ClientServiceProvider.SelectSet**.
- Select set can only contain persisted objects. Cannot add transient objects or objects which are not yet committed.
- To get notified as objects are selected / deselected, use SelectSet events (details next).
- Select Set service maintains hilite of the selected objects.
- Color and Weight are controlled through the GraphicViewMgr

SelectSet …

- SelectSet service offers below properties and functionality.

Properties/Methods

SelectedObjects → BOCollection, which offers below methods/properties

Clear( ) → Clears select set [we will learn about BOCollections soon]

Remove ( ) → removes an object/collection from SelectSet. Has overload variants.

Add ( ) → adds an object/collection to SelectSet. Has overloaded variants.

Clone ( ) → gets a copy of the selected objects as a BOCollection

WaitForUpdate → lets one control incremental display update as elements are added/removed to SelectSet.

- set to **True** to stop notifying.
- set to **False** to notify or start notifying.

Note: those affect only Display, normal notifications are not affected!

Events

Added → raised when an object is added to select set.

Removed → raised when an object is removed from select set.

How to Listen to SelectSet Events

Add a module (Form/Class) level Private *WithEvents* variable of Type SelectSet.

```
Private WithEvents m_oSelectSetEV as SelectSet
```

Implement your logic in m_oSelectSetEV_Added( ) and m_oSelectSetEV_Removed( ) event subroutines to respond to Selection change events.

Initialize it to ***ClientServiceProvider.SelectSet*** when you want to subscribe to SelectSet events.

```
m_oSelectSetEV = ClientServiceProvider.SelectSet ' → subscribed to events
```

Set it to ***Nothing*** when you want to unsubscribe to SelectSet events.

```
m_oSelectSetEV = Nothing ' → unsubscribed from events
```

Note : Select command provides interactive manipulation of the Select Set. Select command cannot run in parallel while another command is running (because only one command can be active). We will learn about Step commands later, which provide mechanism to locate and select objects required for a command's functionality.

TransactionManager

- TransactionManager service provides programmatic access to Transaction Management in Client Tier.
- In simple terms, a Transaction is something which involves Creating/Modifying/Deleting one or more objects. (More details next)
- Accessible as **ClientServiceProvider.TransactionMgr**.
- Provides means to **Commit**, **Abort** and **Compute** *changes** made to BusinessObjects or their properties.
- Changes could be → Create / Modify / Delete / Change properties / Geometric properties like position/orientation.
- Note that S3D has a *Associative System* which reacts to "object modifications" and does find out all the related objects which need to implicitly change. It figures this out from Metadata and Semantics associated with the modified objects (interfaces). So, your transaction may include more objects than just what you have modified.

Transactions

- A transaction is always active when you have accessed any object. i.e. transaction always **begins implicitly**.
- A transaction is always **ended explicitly** by a call to Commit or Abort.
- An S3D transaction could potentially contain many database actions(updates, deletes, inserts). The system determines all implied changes without user's programming intervention. User just modifies the objects he wants.
- No concept of nested transactions. It is one transaction from previous Transaction's end to current state. All changes (explicit changes by user, implicit changes by system) bundle into one S3D transaction. Either all are committed, or all are aborted.
- If system encounters any object which it cannot update, then the transaction ***may*** fail and abort. System tolerates some situations and add to TODO list/display message and proceed to update remaining objects and keep the changes – once again, the object data is always in a valid state.
- Commands are not a logical transaction boundary. Typically
  - Command should NOT "inherit" uncommitted changes of other commands – must do an Abort in its Start.
  - Command should NOT "leave" uncommitted changes it made. Either Abort or Commit before it ends.
- Always try to keep it (Transaction size) small. S3D is based on a data safe ***short-transaction-optimistic-concurrency*** model.

TransactionMgr . Commit or Abort

Commit :

- Flushes out the changes done (since last commit/abort) to the database.
- Takes a string as the undo marker for this operation. Passing empty string means no undo possible for that commit operation.
  - Undo string appears on the Edit > Undo "undo string" menu item, so it is worth considering the "undo string" for internationalization.
- Could throw an exception, be aware and handle (catch) exceptions.
- Could add objects to TODO list if there were some tolerable issues .

Abort :

- Cancels changes done to business objects (since last commit/abort).
- Database not updated.
- All business objects at users hand in logic will revert to their previous state.

TransactionMgr . Commit or Abort …

Releasing Object References :

- There is no need to release objects references (set to Nothing) before Commit/Abort.
- System takes care of releasing references internally across a commit/abort and gives back access to your object variables whenever you access them after the transaction commit/abort.

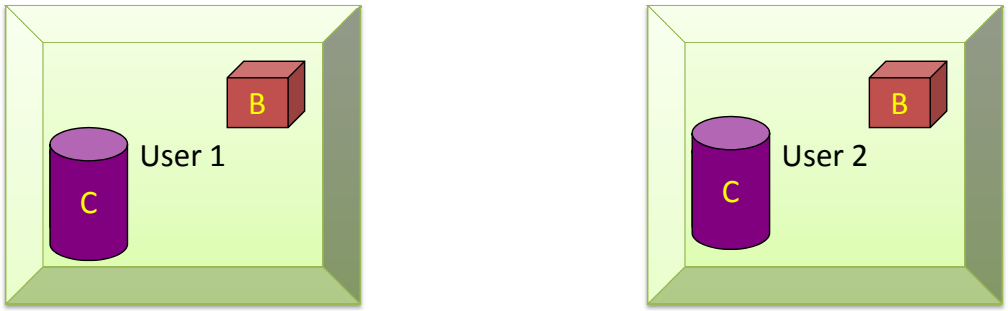
- This is a **BIG benefit** over COM, where you had to set object references to Nothing before a Commit/Abort.

TransactionMgr . Compute

Compute:

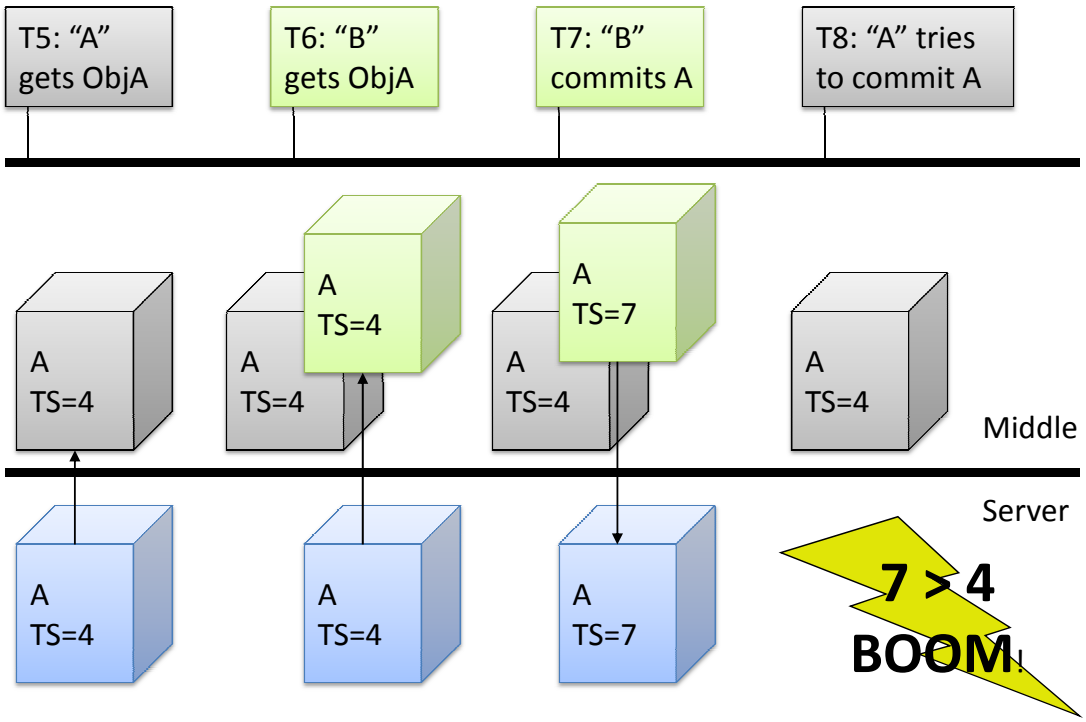
- evaluates all the changes done so far to objects since last Commit / Abort / Compute. It percolates the changes to any dependent objects which the system determines. Eg, deleting a system will percolate down and delete all nested systems and objects.
- invokes the semantics / solvers which react to the modifications made to objects. Solvers & Semantics bring the system to solved state for the modifications made.
- is required before you access any information from related objects for further processing. Eg, changing a Run's diameter would change the diameter of its feature. Hence one should do a compute to access the properties of the modified Run's features.
- updates Display – modified objects show in rubberband hilite mode (dynamics).
- could throw an exception, be aware and handle (catch) exceptions.

Concurrent Users Example



User 1 creates and saves Block B; it appears in Server Tier
User 2 logs on, sees Block B, Can C
User 2 Modifies Can C, but User 1 doesn't see the change yet
User 1 refreshes his display; suddenly sees how Can C has changed!

Timestamping – Concurrent Update handling.



GraphicViewManager

- **GraphicViewManager** service provides programmatic access to Graphic Views, Events, Hiliter etc.
- Accessible as **ClientServiceProvider.GraphicViewMgr**.
- Provides access to
  - Active GraphicView (via the ActiveView property)
  - Create a Hiliter (via the CreateHiliter method)
  - SelectSetColor & SelectSetWeight properties
  - GraphicView events (details next)

GraphicViewManager Events

- You need to use GraphicViewManager events only in these situations
  - You need more events than the basic overridable events in BaseGraphicCommand (We will learn later about Graphic Commands)
  - You need to listen to GraphicViewManager events in a command Form's logic.
- GraphicViewManager funnels events from each individual GraphicView.
  - No need to listen to events from each GraphicView separately.
  - The event signature contains the GraphicView argument which indicates the view where it is coming from.
- GraphicViewEventArgs contains the event info (which button, where etc).
- Event Signature – Click (*oGraphicView*, eGraphicViewEventArgs)

Events	
Click	MouseDown
DblClick	MouseUp
KeyDown	MouseMove
KeyPress	MouseWheel
KeyUp	

Event Information Properties

Alt, Control, Shift → Key states of Alt, Shift and Ctrl keys – True/False

Button → Which Mouse Button was pressed

Delta → Mouse Wheel Roll value

X , Y → Mouse X, Y location From Top-left of View

How to Listen to GraphicViewManager Events

Add a module (Form/Class) level Private *WithEvents* variable of Type GraphicViewManager.

```
Private WithEvents m_oGfxViewMgrEV as GraphicViewManager
```

Implement your logic in the desired event subroutines

```
m_oGfxViewMgrEV_Click( ),
m_oGfxViewMgrEV_MouseDown( ),
m_oGfxViewMgrEV_KeyUp( )
…
```

Initialize it to ***ClientServiceProvider.GraphicViewMgr*** when you want to subscribe to GraphicViewManager events.

```
m_oGfxViewMgrEV  = ClientServiceProvider.GraphicViewMgr ' → subscribed to events
```

Set it to ***Nothing*** when you want to unsubscribe to GraphicViewManager events.

```
m_oGfxViewMgrEV = Nothing ' → unsubscribed from events
```

Very important to unsubscribe otherwise it might keep on receiving events when you really do not intend it to get events. This applies for all Services/Components providing events.

WorkingSet

- **WorkingSet** service represents the set of objects currently in the defined workspace.
- Accessible as **ClientServiceProvider.WorkingSet**
- Provides functionality to
  - Get Active Condition ID → Active user's active permission group
  - Get Active Connection.
  - Get all Connections associated with the working set.
  - Provide ability to do a workspace refresh.

WorkingSet Methods & Properties

WorkingSet Service offers the below methods and properties

- ActiveConditionID property → the permission group ID of the active user
- ActiveConnection property → the active connection to the Plant's database connections, could be Model or Catalog depending on what the active task is.
- Connections property → the set of connections currently fetching data into the current workspace.
- Refresh method → performs an incremental refresh since the last refresh.

ValueManager

- **ValueManager** service provides programmatic access to store and retrieve adhoc values within an active session, mainly useful for commands, client services and components to notify each other of changes without interlinking them in code.
- Accessible as **ClientServiceProvider.ValueMgr**
- Provides functionality to
  - Add, Remove, Replace, Get a Value (Boolean, Double, Integer, String, Object) based on a key (string).
- Raises events
  - ValueChanged
  - KeyAdded
  - KeyRemoved

ValueManager Events, Methods

- ValueManager Service offers the below methods and events

Events	Methods
ValueChanged (key, Object) KeyAdded (Key, Object) KeyRemoved (Key, Object)	Add (Key, Value) → Has overloads for all datatypes Replace (Key, Value) → Has overloads for all datatypes Get<Type>Value as <Type> → for all datatypes GetObjectValue as Object → for Object datatypes Remove (Key) → Removes a value by key IsKeyValid (Key) → tells whether a key is valid or not. Count → Returns Count of Keys GetKeyName (index) → gets key of value by index Datatypes → (Boolean, Double, Integer, String)

How to Listen to ValueManager Events

Add a module (Form/Class) level Private *WithEvents* variable of Type ValueManager.

```
Private WithEvents m_oValueMgrEV as ValueManager
```

Implement your logic in the desired event subroutines

```
m_oValueMgrEV_KeyAdded(Key as string, value as Object),
m_oValueMgrEV_KeyRemoved(Key as string, value as Object),
m_oValueMgrEV_ValueChanged (Key as string, value as Object),
…
```

Initialize it to ***ClientServiceProvider.ValueMgr*** when you want to subscribe to ValueManager events.

```
m_oValueMgrEV = ClientServiceProvider.ValueMgr ' → subscribed to events
```

Set it to ***Nothing*** when you want to unsubscribe to ValueManager events.

```
m_oValueMgrEV = Nothing ' → unsubscribed from events
```

Preferences Service

- **Preferences** service provides programmatic access to store and retrieve adhoc values across sessions (ie. into Session File).
- Typically used to store “last used values” by commands. For example, last used Pipeline for Routing, Last used Section Name for Structural Member placement, Last picked equipment for placement and so on.
- Accessible as **ClientServiceProvider.Preferences**
- Provides functionality to
  - Get / Set a Value (Boolean, Double, Integer, String) based on a key (string).
- Raises events
  - ValueChanged

Preferences Events, Methods

- Preferences Service offers the below methods and events

Events	Methods
ValueChanged (key, Object)	SetValue (Key, Value) → Has overloads for all datatypes Get<*Type*>Value as <*Type*> → for all datatypes Count → Returns Count of Keys Datatypes → (Boolean, Double, Integer, String)

How to Listen to Preferences Events

Add a module (Form/Class) level Private *WithEvents* variable of Type Preferences.

```
Private WithEvents m_oPreferencesEV as Preferences
```

Implement your logic in the desired event subroutines

```
m_oPreferencesEV_ValueChanged (Key as string, value as Object),
…
```

Initialize it to ***ClientServiceProvider.Preferences*** when you want to subscribe to Preferences service events.

```
m_oPreferencesEV = ClientServiceProvider.Preferences ' → subscribed to events
```

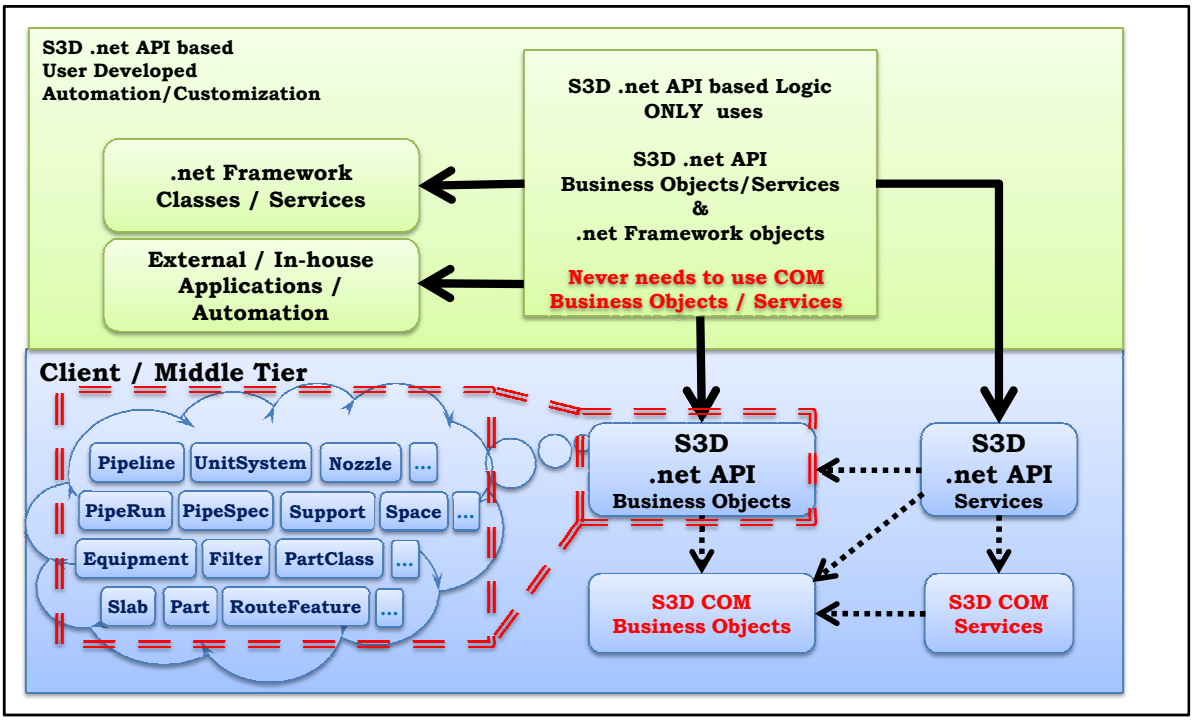
Set it to ***Nothing*** when you want to unsubscribe to Preferences Service events.

```
m_oPreferencesEV = Nothing ' → unsubscribed from events
```

S3D .net API

Basics of Business Objects

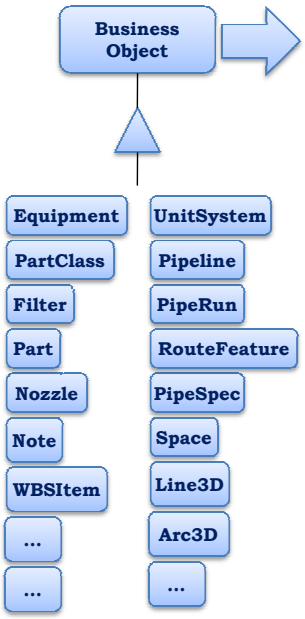
Business Objects



Business Objects

- Abbreviated as ***BO***. BusinessObjects are the engineering things like Equipment, PipeRun, PipeFeature, ControlPoint and so on.
- A .net BusinessObject is also referred to as ***Wrapper***, as it essentially wraps the Business Object and provides property/method access.
- For people who know S3D COM object model, incidentally, this also wraps the internal COM business object. The internal COM business Object is never available, nor needed for end user's usage in S3D .net API.
- Provides many useful methods and properties, including generic access to properties and relationships. (more on this in later session).
- The Basic Business object (BusinessObject) is the base class of all S3D Business objects in .net API. It inherits from System.Object.
- Each S3D business object is wrapped by a .net BusinessObject. There could be a specialized BO wrapper which offers more methods & functionality, like a PipeRun or a Equipment Instance or a Catalog Part. Wherever there is no Specialized BO Wrapper, the basic BusinessObject provides the wrapper.

Business Objects



AccessControl	Gets AccessControl to current user
ApprovalStatus	Gets ApprovalStatus of the object
ClassInfo	Gets ClassInfo (Metadata) of this object
DBConnection	Gets Connection to which this BO belongs.
Delete()	Deletes the Object (if permissions allow)
GetAllProperties	Gets PropertyValue ReadOnlyCollection of all properties
GetPropertyValue()	Gets PropertyValue given Property Details
SetPropertyValue()	Sets PropertyValue given Property Details and value
GetRelationship ()	Gets RelationshipCollection given RelationshipInfo.
IsTypeofBOC ()	Checks if this BO is of given BO Classification type.
Notes ()	Gets Notes associated with this BO (as ReadOnlyCollection).
RemoveAllNotes ()	Removes all Notes associated with this BO.
RemoveNote ()	Removes a given Note object associated with this BO.
ObjectID	Gets OID of the Object
PermissionGroup	Get/Set the PermissionGroup to which this object belongs
Relationships ()	Gets read only collection of RelationCollection.
SupportsInterface()	Check if the BO supports a given interface name
ToString()	Gets name if BO is IJNamedItem, OID if not Named Item.
UserClassInfo()	Gets ClassInfo (Metadata) for user bulkloaded partclass occs

Business Objects

- As you can see, most of the information needed about a Business object is available right at this level, and it serves such information for any kind of S3D object. For example…
  - An object's Object type, i.e. Class Information; Implemented Interfaces; Its BOC classification; AccessControl to current user; whether or not it supports an interface; its permission group; status; the connection to which it belongs; property access – Get/Set; its ID in the database; its Related objects; its User Class Information, its Notes etc.
- Some more information is just one step away, for example, to get related objects given a relationship & Role names is just one step away… (we will learn about Relationships and Related objects in a later session)
- For people who know S3D COM object model, all this information was scattered over different interfaces – which was so to take benefit of the COM methodology of partitioning data & information on various interfaces.
- It is a great advantage in **S3D .net API** to have easy access to all such important information. This greatly reduces the number of lines you have to write to perform common tasks like get/set properties, access relations, metadata, access control and many more.

S3D .net API

Collections

Collections

- Several times client tier programming deals with collections of business objects.
- For example,
  - Filter would return a collection of objects
  - Selected Objects are made available as a collection by SelectSet Service
- In S3D .net API we have these collection types
  - BOCollection → Business Object Collection.
  - ReadOnlyBOCollection → non-modifiable Business Object Collection
  - ReadOnlyDictionary
  - ReadOnlyCollection (System.Collections.ObjectModel.ReadOnlyCollection)

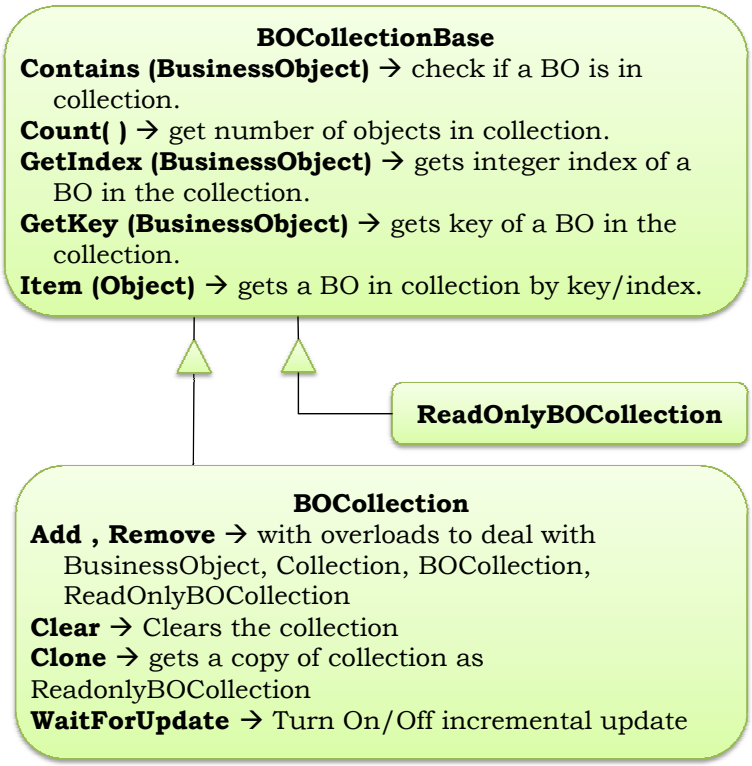
We will know more about these in this session

BOCollection classes

BOCollectionBase has those methods and properties which are common for BOCollection and ReadOnlyBOCollection - access Count, Members(BOs), Get Key/Index of a given BO, check if collection contains a BO etc…

ReadOnlyBOCollection has no more methods than the base class. No modifications to collection possible.

BOCollection offers methods to modify Collection, i.e. Add/Remove/Clear and a method to clone the collection, and also to control how incremental updates to collection are broadcasted.



ReadOnlyDictionary

- ReadOnlyDictionary is a generic collection class maintaining a collection of objects as a key/value pair.
- Has methods / properties to access Count, Get Keys/Values as collection, check if a key or key/value pair exists, get Value by Key etc.

ReadOnlyDictionary

Contains (Key,Value) → check existence of key/value pair.
ContainsKey (Key) → check existence by key.
Count( ) → get number of objects in dictionary.
Keys ( ) → access Keys in the dictionary.
Values ( ) → gets Values in the dictionary.
TryGetValue(key,Object) → gets a value by key.

- Most Metadata related API methods/properties return this kind of collection.

Properties / Methods returning Collections

- API methods or properties returning Collections
  - **Filter.Apply** → returns filtered objects collection
  - **SelectSet.SelectedObjects** → the collection of Selected objects.
  - **Locator.LocatedObjects** returns the located objects
  - **Hiliter.HilitedObjects** maintains the hilited object collection.
  - **Metadata** methods return ReadOnlyDictionary collections for BOCs, Classes, Codelists, Properties, etc.
  - Similarly, there are several instances where methods/properties return back collections of one of the four kinds we saw here.

S3D .net API

LAB

-

Manipulating Select Set

(Using Client Services, Dealing with Collections, Events etc)

LAB - Overview

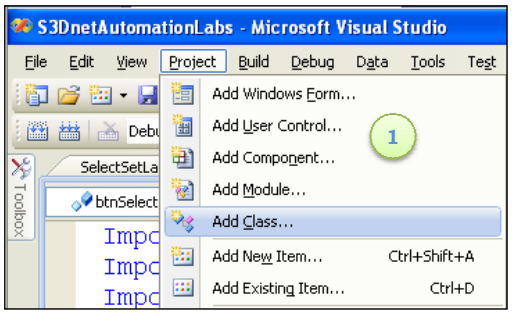
- In this LAB we will learn
  - Writing a Custom command in VB.net which manipulates SelectSet.
    - Access SelectSet Service
    - Access Objects in SelectSet
    - Dealing with BOCollection
    - Remove Objects from SelectSet
    - Check if SelectSet contains an Object
    - Add Objects to SelectSet
    - Clear Objects in SelectSet
    - Use DelayedUpdate while manipulating SelectSet
    - Listen to SelectSet events

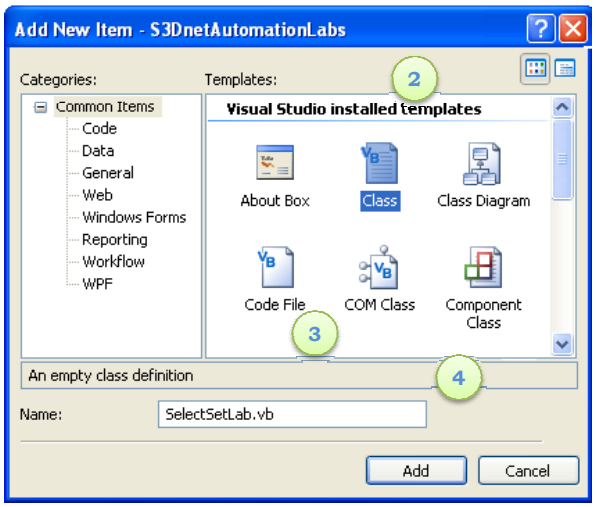
Create a new Command Class

1. Inside the Visual Studio Project, Click Project > Add Class… menu item.
2. Select **Class** from the Templates.
3. Specify the Name of the class file as desired.

We choose “SelectSetLab.vb” for name as we intend to create the new class name as “SelectSetLab”.

4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.





Create a new Form for Command

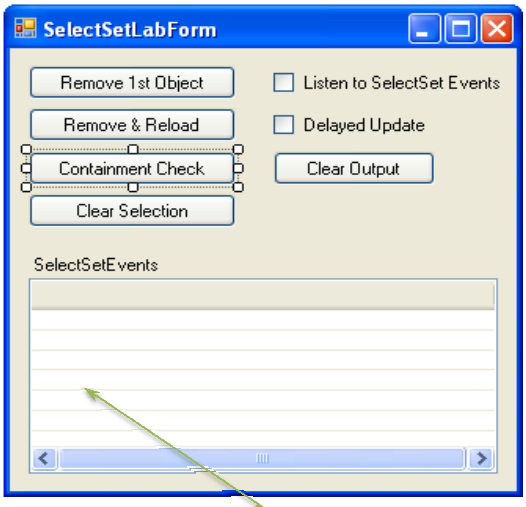
Our SelectSetLab command needs a form as we need to do some UI interaction.

Create and add a form to the project.

1. Inside the Visual Studio Project, Click Project > Add Windows Form … menu item.
2. Select **Windows Form** from the Templates.
3. Specify the Name of the Form file as desired.

We choose “SelectSetLabForm.vb” for name as we intend to create the new form for our “SelectSetLab” command.

4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.



ListView : Name= lvEvents,
View = Details
Add a **Column** named “Events” to the **Columns** property of this ListView

Implementation

Our SelectSetLab command will Inherit from BaseModalCommand and has a form.

It will be a Modal command, so we return **True** for **Modal** property.

In OnStart( ), we instantiate the Form reference and show it.

No other Override implementation is necessary in the Command.

```
Public Class SelectSetLab
  Inherits BaseModalCommand

  Public Overrides Sub OnStart(ByVal co
    Dim oForm As frmSelectSetLab

    'Create form
    oForm = New frmSelectSetLab

    'show as Modal form
    oForm.ShowDialog()

  End Sub

End Class
```


Implementation in the Form

First add the import statements at the top of the form's code

Our SelectSetLabForm needs a SelectSet and also has an events variable of SelectSet type.

On **Form Load**, we get SelectSet service.

On **btnRemove1stObject's Click**, we call .SelectedObjects.Remove method passing the 0th object.

On the **'btnClearSelection' click**, we call .SelectedObjects.Clear

On the **'chkDelayedUpdate' check/uncheck**, we set WaitForUpdate flag to true/false accordingly.

```
'Declare a SelectSet Service reference
Private m_oSelectSet As SelectSet
'Declare a WithEvents SelectSet Service
'Service to listen to SelectSet events
Private WithEvents m_oEVSelectSet As SelectSet
```

```
' Get Selectset service
m_oSelectSet = ClientServiceProvider.SelectSet
```

```
m_oSelectSet.SelectedObjects.Remove(
    m_oSelectSet.SelectedObjects.Item(0))
```

```
' Clear Selected Objects
m_oSelectSet.SelectedObjects.Clear()
```

```
' Set the WaitForUpdate flag on Selected Objects
m_oSelectSet.SelectedObjects.WaitForUpdate
    = chkDelayedUpdate.Checked
```

Implementation in the Form…

On the 'chkListenToEvents' check/uncheck, we accordingly initialize/release the SelectSet events variable and log it to our events list view.

```
If (chkListenToEvents.Checked = True) Then
    ' Set our Events variable to the SelectSet service, Starts listening
    m_oEVSelectSet = m_oSelectSet
    lvEvents.Items.Add(" Started Listening to SelectSet Events")
Else
    ' Set our Events variable to Nothing, i.e. stops listening .
    m_oEVSelectSet = Nothing
    lvEvents.Items.Add(" Stopped Listening to SelectSet Events")
End If
```

On the **Added**, **Removed** events of m_oEVSelectSet, we report BO's name to lvEvents Listview. In Form's Code window, pick m_oEVSelectSet on the LHS Controls Combobox, and the RHS events list has the Added & Removed events.


```
Private Sub m_oEVSelectSet_Added(ByVal BusinessObject As Ingr.S
    lvEvents.Items.Add("Added " & BusinessObject.ToString)

Private Sub m_oEVSelectSet_Removed(ByVal BusinessObject As Ingr
    lvEvents.Items.Add("Removed " & BusinessObject.ToString)
```


Implementation in the Form…

On 'btnRemoveAndReloadAll' click,

- We Clone the SelectedObjects collection for further use below.
- Remove the Cloned Object Collection from SelectedObjects.
- Observe Hilite not updated if WaitForUpdate = True. Setting it to False Updates Hilite.
- Add the objects back, one-by-one from saved Collection, and finally do a Force Update. Observe that
  - if DelayedUpdate = True → process is fast & no flicker. Hilite not updated incrementally.
  - if DelayedUpdate = False → process is slow & flickery. Hilite updated incrementally.

```
' Get a Copy of Selected Objects
Dim oSelObjs As ReadOnlyBOCollection
   = m_oSelectSet.SelectedObjects.Clone
```

```
' Remove those selected Objects.
m_oSelectSet.SelectedObjects.Remove(oSelObjs)
```

```
' Hilite isnt updated yet if DelayedUpdate is ON.
' Force it.
If (chkDelayedUpdate.Checked) Then
    With m_oSelectSet.SelectedObjects
        .WaitForUpdate = False ' Hilite Updates
        'Revert to setting on the form.
        .WaitForUpdate = chkDelayedUpdate.Checked
    End With
End If
```

```
' Add back originally Selected objects one-by-one
For Each oBO As BusinessObject In oSelObjs
    ' Add the object to Select set
    m_oSelectSet.SelectedObjects.Add(oBO)
Next
' Can also add entire collection at once, as below
'm_oSelectSet.SelectedObjects.Add(oSelObjs)
```

Also Insert the above Hilite Update logic here as well after adding the objects one-by-one.
We want display updated after adding all objects

Copy

Implementation in the Form…

On **'btnContainmentCheck' click.**

- We first get the 0th object in the SelectedObjects as **oBO**.

```
Dim oBO As BusinessObject = _
       m_oSelectSet.SelectedObjects.Item(0)
```

- Check if it exists in SelectedObjects by using SelectedObjects.Contains (**oBO**). It returns **True** as it exists.

```
MsgBox("SelectSet Contains Object(0) Originally? " _
       & m_oSelectSet.SelectedObjects.Contains(oBO))
```

- We remove **oBO** from SelectedObjects and Check again. It returns **False** as it doesn't exist now.

```
m_oSelectSet.SelectedObjects.Remove(oBO)
MsgBox("SelectSet Contains Object(0) After Remove(0)?" _
       & m_oSelectSet.SelectedObjects.Contains(oBO))
```

- We Add Back **oBO** from SelectedObjects and Check again. It returns **True** as it exists now.

```
m_oSelectSet.SelectedObjects.Add(oBO)
MsgBox("SelectSet Contains Object(0) After Added Back?" _
       & m_oSelectSet.SelectedObjects.Contains(oBO))
```


Implementation in the Form…

On '**Form_Closed' event**,

- we unsubscribe from SelectSet events by setting the Events variable to Nothing. Otherwise, we risk the danger of the system trying to notify the Commands form upon Process exit as it had no clue that the listener reference has not been set to nothing with the system's notice.

```
    Private Sub frmSelectSetLab_FormClosed(ByVal
        m_oEVSelectSet = Nothing
    End Sub
```

- we need to follow the same model with all other EVENT variables, eg GraphicViewMgr etc.
- The remaining trivial implementation of ClearOutput button is as below.

```
Private Sub btnClearOut_Click(ByVal sender A
  lvEvents.Items.Clear()
End Sub
```

Our command implementation is complete now, and you can Run/Debug the command.

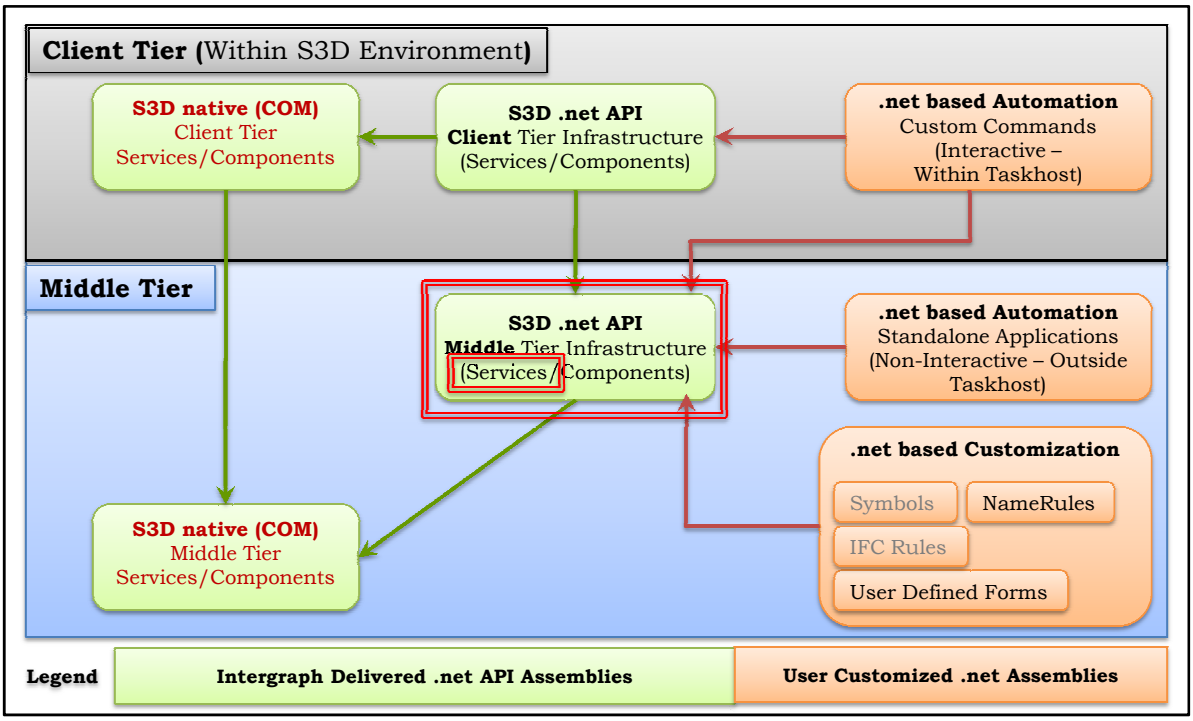
S3D .net API

Middle services

-

Automation Services
unrelated to User Interface

S3D .net API – Middle Services

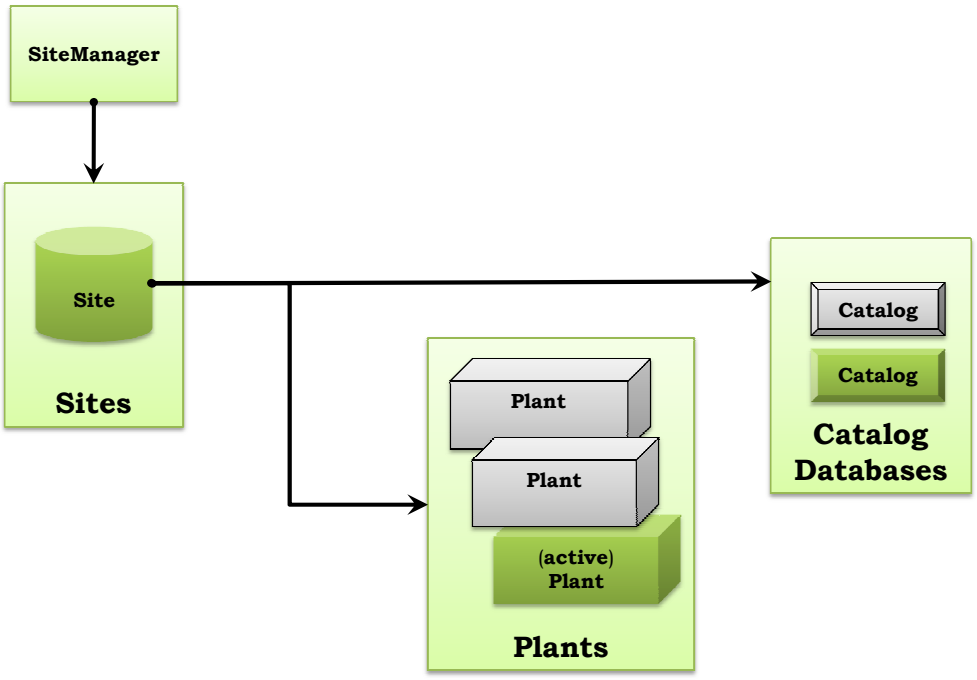


CommonMiddle Services

- *MiddleServiceProvider*, a *static* class, provides access to needed services in the Middle Tier, which in-turn provide access to components and other artifacts useful both in Middle and Client Tiers.
  - SiteMgr
  - UOMMgr
  - TransactionMgr
  - ErrorLogger
  - ActiveSite, Plants, Catalogs
  - ActivePlant, its Catalog & Schema
  - Connection Information (Site, Location, Model, Catalog, Reports, References)
  - Metadata Information via MetadataManager – Classes, Interfaces, Relations, Properties, Codelists, BOC, Edges

Available in **CommonMiddle.DLL**, namespace : **Ingr.Common.Middle.Services.**
Being a static class, **MiddleServiceProvider** can be used directly without the need to **new** it.
Never use **New** operator to get a Middle service. Always get it from **MiddleServiceProvider**.

SiteManager, Site, Plants, Catalogs …



SiteManager

- **Sites()** → Collection of Sites.
  - Today S3D knows one Site, the one configured using "Modify Database and Schema Location" utility.
  - This property is provided as collection to enable future access to all available sites…
- **ConnectSite ( )** → Method to connect to a Site.
  - Useful in Standalone Apps. Cannot use in Interactive Commands as active Site is already is activated when inside S3D Taskhost.
- **ActiveSite** property
- **SiteServerName** property

Eventually from the SiteManager and its above methods/properties, one can get to all Plants, Catalogs, Activate a Plant, get its Catalog, Model, Reports, References connections etc.

We discuss all these separately in a later session.

MetadataManager

- Access from Model, Catalog, Report and References.
  - Like Model.MetadataMgr
- Provides Metadata information
  - Classes, Interfaces, Properties, Codelists, Relationships, BOC, Edges
- Available as
  - Collections (ReadOnlyDictionary (Of <*Type*>Information)
  - Methods to get individual item: Get<*Type*>Info

eg,

- ReadOnlyCollection<ClassInformation> **Classes**
  ClassInformation **GetClassInfo**(string className)
- ReadOnlyCollection<InterfaceInformation> **Interfaces**
  InterfaceInformation **GetInterfaceInfo**(string interfaceName, string nameSpace)

(Metadata API is covered fully in a separate session later)

UOMManager

- Access as MiddleServiceProvider.UOMMgr – Provides Units management, Parsing input values from user, Formatting internal values for display etc.
  - Parse a Unitted string input to corresponding numeric value.
    - → ParseUnit(unitType, inputString) as Double ' interpret input value string for a unitType
  - Format a property value to corresponding unitted displayable readout value.
    - → FormatUnit(propertyValueDouble, …) as string ' from propertyValueDouble
    - → FormatUnit(unitType, Double, …) as string ' from Value and unitType
  - Convert DataBase value of a property to its active units, or a given unit.
    - → ConvertDBUtoUnit(propertyValueDouble, …) as Double ' from PropertyValueDouble
    - → ConvertDBUtoUnit(unitType, Double) as Double ' from Value and unitType
  - Convert a unitted value to Database Units.
    - → ConvertUnitToDBU(unitType, Double) as Double ' from Default active units
    - → ConvertUnitToDBU(unitType, Double, unitName) as Double ' from a Given unit.
  - Get/Set active units for a given unit type.
    - → GetDefaultPrimaryUnit( ), GetDefaultSecondaryUnit( ), GetDefaultTertiaryUnit( ), SetDefaultUnits( )
  - Get/Set active unit format for a given unit type.
    - → GetDefaultUnitFormat(unitType), SetDefaultUnitFormat(unitType, UOMFormat) ' Readout control : precisionType, Decimal/Fractional precision, Leading/Trailing zeros, ReduceFraction, UnitsDisplayed or not.

TransactionManager (in Middle Tier)

- TransactionManager service **in Middle Tier** provides programmatic access to Transaction Management when desired in Middle Tier.
- Accessible as **MiddleServiceProvider.TransactionMgr**.
- For transaction control (Commit/Abort/Compute) – Use this only in Middle Tier. Use the ClientServiceProvider.TransactionManager for all other interactive usage cases within Taskhost.
- Provides means to **Compute, Commit**, **Abort** *changes** made to BusinessObjects or their properties.
- Changes could be → Create / Modify / Delete / Change properties / Geometric properties like position/orientation.
- All other basic information/concepts about transactions holds good as detailed in the Client Services session.

ErrorLogger

- ErrorLogger service provides means to log errors at runtime as experienced by a client tier/middle tier code.
- Accessible as **MiddleServiceProvider.ErrorLogger**.
- Offers methods to Log an error (with overload variants to provide more details about the error). Also offers properties to set ErrorLogging state (On or Off), and Specify a LogFile.
  - ErrorLogger.Log(errorString) is the primary method one could use to log an error. Couple of more overloads exist, which take more details about the error and log it.
- Even though properties exist to control logging, it is best to leave them to the active defaults set by ErrorLogEnable.exe
  - ErrorLogger.EnableLog( ) = True ‘ or False as desired.
  - ErrorLogger.PersistFilePath → set the full path to the Error Log filename.

S3D .net API

Graphic Commands

Graphic Commands

- Inherit from BaseGraphicCommand, and, Typically
  - are non-modal and suspendable most of the time.
  - their **OnStart( )** → sets up things (Fence, Locator(s), Hiliter(s) etc), interprets object inputs from SelectSet, initializes and shows its Form / RibbonBar (if any) and subscribes to GraphicView Manager events.
  - their **OnSuspend( )** → Hides its Form(s) if any, un-subscribes to events.
  - their **OnResume( )** → UnHides its Form(s) if any, re-subscribes to events.
  - their **OnStop( )** → has cleanup code. Un-subscribe from events, closes its Form(s) if any.
  - their **OnIdle ( )** → Rarely used. Can implement any idle time processing.
  - Provide a Prompt / error message to the user in the Status Bar using **WriteStatusBarMsg**
  - On Mouse events, gets 3D world position of 2D view position using **XformScreenToWorld** method, and do an action (discussed next)
  - On RibbonBar events, acts on the inputs / actions from the RibbonBar

Graphic Commands …

- **OnMouseDown → Creates** Business object(s) as per the command's functionality spec and issues a Transaction Manager **Compute**.
- **OnMouseMove → interprets** the mouse position and **modifies** the Business object(s) properties as per the command's functionality spec and issues a Transaction Mgr **Compute**.
- **OnMouseUp → interprets** the mouse position and changes the Business Object(s) properties as per the command's functionality spec, issues a Transaction Mgr **Compute**, and then a **Commit**. Command may continue further to create/modify the next object as per command's functionality spec.
- **OnKeyDown →** has implementation which reacts to key down event. Usual behaviour is to **Stop** the command on an **Escape** KeyDown event. Behaviour for other keys is as specified by the command's functionality spec.
- **OnKeyUp →** has implementation which **reacts** to key up event. Behaviour is as specified by the command's functionality spec.
- **RibbonBar Events →** has specific implementation as specified by the command's functionality spec.

Graphic Commands… implementing more events

- Note that you don't have *direct* means to implement the remaining other Mouse (eg Click, Roll) and Keyboard events *directly* by overriding the Graphic Command Base Class implementation.
- If you need to implement those events, you would do the following.
  - Define a Class level private WithEvents variable of type GraphicViewManager, lets say we call it m_oEVGfxVwMgr. (EV stands for Events Variable)
  - In **OnStart** and **OnResume** implementation, initialize it (m_oEVGfxVwMgr) to ClientServiceProvider.GraphicViewMgr
  - Inside **OnSuspend** and **OnStop**, set it (m_oEVGfxVwMgr) to Nothing
  - Implement the necessary events on m_oEVGfxVwMgr.
  - Note that you don't want to end up doing a double implementation (for example both GraphicCommand.*On*MouseDown and m_oEVGfxVwMgr_MouseDown)
  - Instead of using a WithEvents variable, you could also do it using "Delegates and AddHandler/RemoveHandler" approach in **.net**. Read more on "Delegates and Event Handling" in **.net** to know how.
- Similarly, if you have another 3rd party automation component you want to use in your command and deal with events from it, or make it listen to Graphic view events, you do it in one of the approaches as explained above.

Graphic Commands… Using Forms & Events

- Graphic Commands can use the ActiveView property to
  - Create Locator(s), Rectangular Fence as needed by the Command's functionality specification.
  - Modify MousePointer, for example to show Cross Hair cursor, or wait cursor.
- Note that ActiveView property keeps changing as the user changes the active view by clicking in another view.
- Graphic Commands can also show forms. For this purpose :
  - add the required form into the project, add the required controls on it,
  - In the Command Class add a WithEvents reference to the form
  - Initialize the form and its controls in Command's OnStart implementation and finally show the form. Use Show(), not ShowDialog().
  - Upon **Form Close**, you would stop the command. For this purpose, you would register for Form events in the Command class and handle the Close event.
  - Also, when the command is stopped by the system, you would add necessary code to close and unload the form.
  - This (Stopping a Command) is a little different than what you could also do in COM (Using CommandManager.StopCommand (commandID)). There is no access to Command Manager in .net (as yet).
  - To stop a Command from within the Command Logic, just call StopCommand( ). This instructs Command Manager to stop this command.

Using RibbonBars in (Graphic/Step) Commands

- Smart3D provides RibbonBar functionality – essentially a command specific UI which blends into the Smart3D UI.
  - Available with Graphic/Step commands.
  - RibbonBar is a UserControl, but is limited in height (41pixels).
  - Controls on it generally deal with the BusinessObjects handled by the command.
  - Be judicious about the controls you use – avoid bigger controls – Buttons, Labels, Text/Combo/Check/Radio boxes etc are good controls to use.
  - If you use licensed controls, you must ensure the licenses to those are available at runtime/design time as appropriate.
  - To use a RibbonBar in a command –
    - Create a UserControl and change it to inherit from BaseRibbonBarControl (control.designer.vb)
    - RibbonBar can expose events/properties to Command and command can listen/get/set them.
    - Use the WithEvents approach described earlier to handle RibbonBar Events.
    - Create RibbonBar instance in **command's OnStart**, initialize its fields / properties
    - set Command's **.RibbonBar** property to the RibbonBar Instance you created.
    - Smart3D takes care of hiding/showing the RibbonBar when command suspends/resumes.
    - Always try to separate the UI and Business logic. Handle the logic in command, and raise events from RibbonBar which the command interprets. Use RibbonBar as a data *presenter*.

S3D .net API

LAB

–

Simple Graphic Command

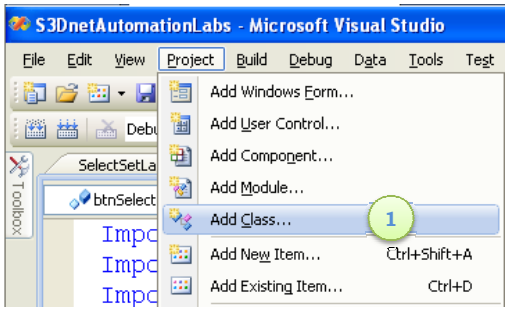
LAB - Overview

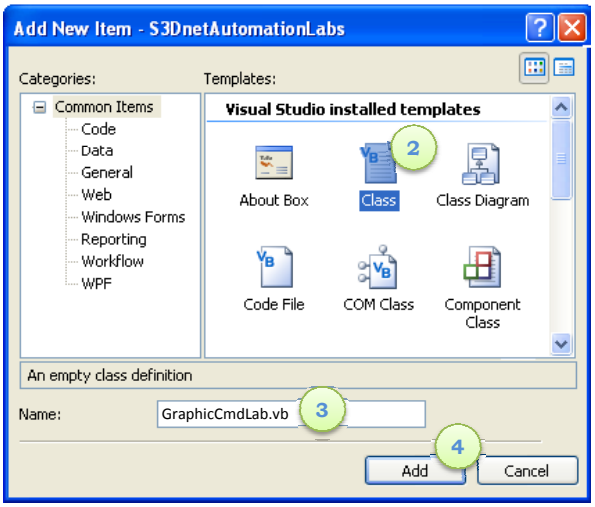
- In this LAB we will learn
  - Writing a Graphic command in VB.net which displays a form showing the View and World coordinates of the mouse pointer.
    - Create a Non-Modal Command.
    - React to Mouse and Keyboard Events.
    - Displaying a Form, show some info on Form.
    - Convert Mouse Coordinates in Graphics View to 3D world coordinates.
    - Stop command when user hits the Escape Key or clicks the Right Mouse button.

Create a new Command Class

1. Inside the Visual Studio Project, Click Project > Add Class… menu item.
2. Select **Class** from the Templates.
3. Specify the Name of the class file as desired.

   We choose “GraphicCmdLab.vb”
4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.

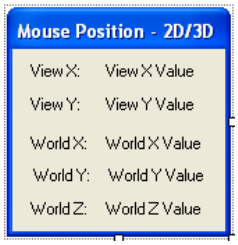




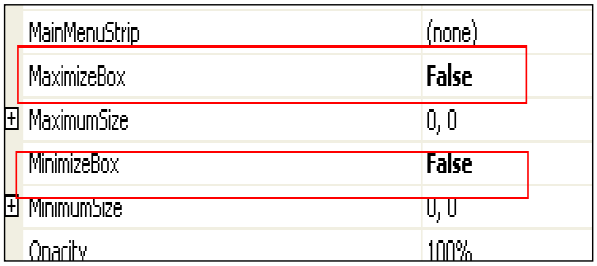
Create a new Form for Command

Our GraphicCmdLab command needs a form as we need to do some UI interaction to show the 3D world Coordinates for the mouse position in the Graphics View.

Create and add a form to the project.



1. Inside the Visual Studio Project, Click Project > Add Windows Form … menu item.
2. Select **Windows Form** from the Templates.
3. Specify the Name of the Form file as “frmGraphicCmdLab.vb”. In the Form Properties window set “MaximizeBox” and “MinimizeBox” to **False**
4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.



Implementation

Our GraphicCmdLab command will Inherit from BaseGraphicCommand and has a form. By inheriting from BaseGraphicCommand our command will be Non-modal and Suspendable by default, so we do not need to override the "Modal" and "Suspendable" properties.

Our Command needs an ActiveView to run, so we override EnableUIFlags property.

In OnStart( ) we instantiate the Form reference and show it as a Non-modal form (**Show()** as opposed to **ShowDialog()**) so that we can interact with Graphic Views, which in-turn will raise graphic view events appropriately.

In OnStop() we close the form and release it.

In OnSuspend() we hide the form, and
In OnResume() we show it again.

```
Public Class GraphicCmdLab
    Inherits BaseGraphicCommand

    Private m_frmCmdForm As frmGraphicCmdLab

    Public Overrides ReadOnly Property EnableUIFlags()
        Get
            Return EnableUIFlagSettings.ActiveView
        End Get
    End Property

    Public Overrides Sub OnStart(ByVal commandID As In
        m_frmCmdForm = New frmGraphicCmdLab
        m_frmCmdForm.Show()
    End Sub

    Public Overrides Sub OnStop()
        m_frmCmdForm.Close()
        m_frmCmdForm = Nothing
    End Sub

    Public Overrides Sub OnSuspend()
        m_frmCmdForm.Hide()
    End Sub

    Public Overrides Sub OnResume()
        m_frmCmdForm.Show()
    End Sub
```


Implementation of Graphic and Keyboard Events

In OnMouseDown() we check if the Right Mouse button was pressed. If yes we stop the command , using the StopCommand method. The below picture is only checking for e.Shift. You will have to check for *e.Shift = 0 And e.Control = 0 and e.Alt = 0*

```
Protected Overrides Sub OnMouseDown(ByVal view As Ingr.SP3D.Common.Client.Services.GraphicVi

    If (e.Button = GraphicViewEventArgs.MouseButtons.Right And e.Shift = 0) Then
        StopCommand()  ' RightClick, must stop the Command.
    End If

End Sub
```

Similarly, in OnKeyDown() we check if the Escape key was pressed, and if yes we stop the command, using the StopCommand method.

```
Protected Overrides Sub OnKeyDown(ByVal e As Ingr

  If e.KeyValue = ConsoleKey.Escape Then
    StopCommand()
  End If
End Sub
```

Implementation of Graphic and Keyboard Events

In OnMouseMove() we get 3D world Position as parameter. We use it and output it on the form along with 2D mouse position.

Note : 3D World Position is not a parameter for **GraphicViewMgr events** (like MouseWheel …) – you can use **XformScreenToWorld** method, passing the View and the 2D mouse Coordinates to get the equivalent 3D world position.

```
Protected Overrides Sub OnMouseMove(ByVal view As I

    m_frmCmdForm.lblViewXValue.Text = e.X
    m_frmCmdForm.lblViewYValue.Text = e.Y
    m_frmCmdForm.lblWorldXValue.Text = position.X
    m_frmCmdForm.lblWorldYValue.Text = position.Y
    m_frmCmdForm.lblWorldZValue.Text = position.Z

    ' 3D world Position available on OnMouseMove
    ' method signature.
    ' GraphicViewMgr Events don't have 3D Position
    ' parameter. Get it like below.

    ' Dim oPos As Position
    ' oPos = XformScreenToWorld(view, e.X, e.Y)

End Sub
```

Our command implementation is complete now, and you can Run/Debug the command.

S3D .net API

LAB

-

Equipment Modification Command

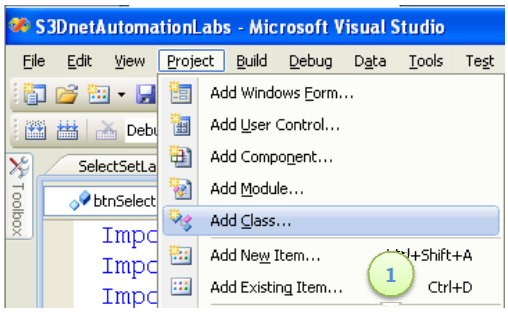
(Modifying Objects, Transaction Management, UOMMgr, Form/Command/Graphics interaction)

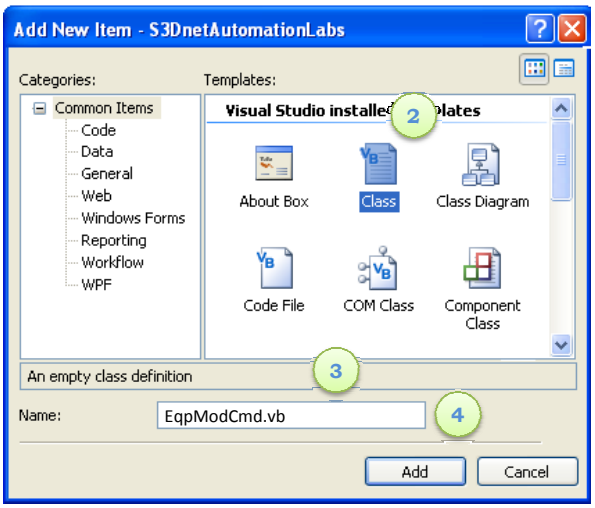
LAB - Overview

- Main purpose in this lab is to show how to modify object(s) [graphic position, a simple direct attribute (eg name)], use UOMmgr to format/parse unitted value outputs/inputs and use transaction manager compute/commit/abort.
- We choose Equipment modification because it is just a easy topic for this. Look at Equipment related functionality in this LAB as a black box. We will learn about Equipment Automation functionality in detail in a later session.
- In this LAB we will learn
  - Writing a Graphic Custom command in VB.net which will display and allow the user to modify both the name and position of an Equipment.
  - Creating a Graphic (Non-Modal, Suspendable) Command.
  - React to Mouse and Keyboard Events from Graphic View.
  - Show a Non-modal form & React to user interaction with the form.
  - Use UOM service to format/parse a unitted value output/input to/from user.
  - Move an Equipment to the Mouse Position.
  - Set Selected Equipment Name.
  - Use Transaction Manager Commit/Compute/Abort.

Create a new Command Class

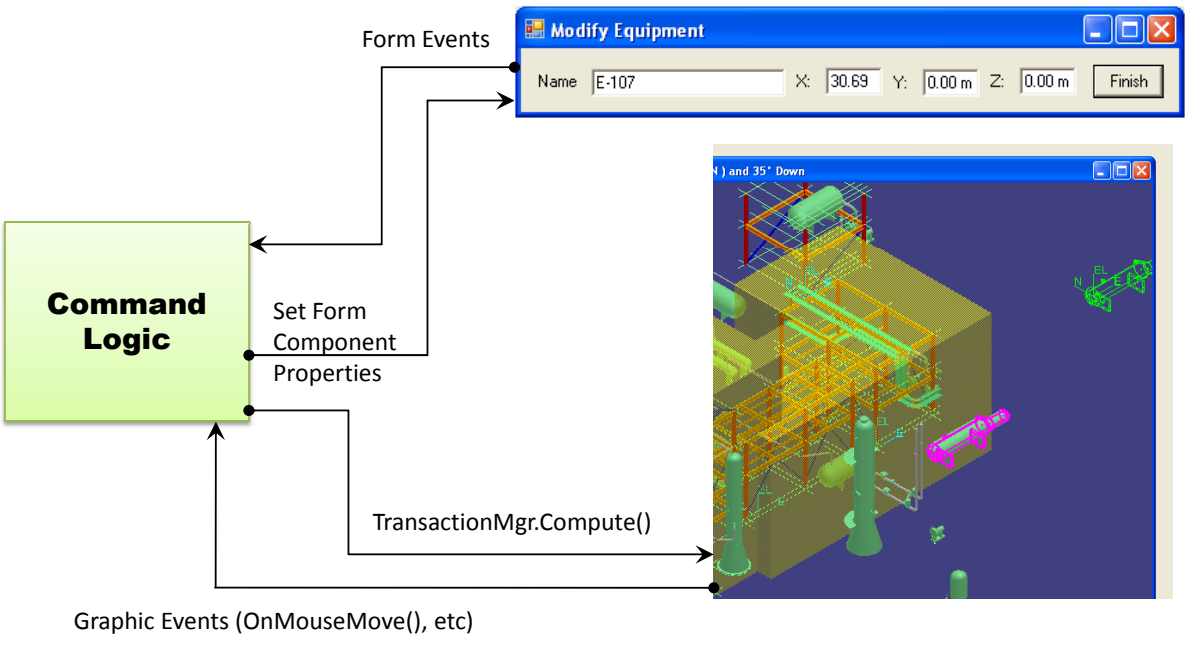
1. Inside the Visual Studio Project, Click Project > Add Class… menu item.
2. Select **Class** from the Templates.
3. Specify the Name of the class file as “EqpModCmd.vb”
4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.





Command and Form (UI) Interaction Design

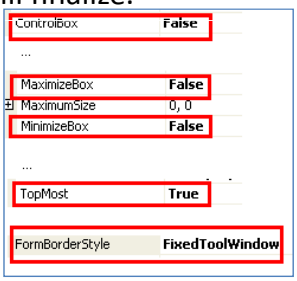
Our command will allow the user to interact with it both through Mouse and Keyboard events and through the form.



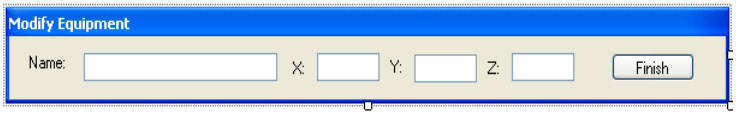
Create a new Form for Command

Our Create a new form named frmEqpModCmd and add it to the project using following steps.

1. Inside the Visual Studio Project, Click **Project > Add Windows Form** … menu item.
2. Select **Windows Form** from the Templates.
3. Specify the Name of the Form file as "frmEqpModCmd.vb" and click Add.
4. A new form file gets created and added to our project. It contains User Interface design and implementation of the Form class which we will finalize.
5. In the Form Properties window set
   "MaximizeBox" , "MinimizeBox", "ControlBox" to **False,**
   "TopMost" to **True, "**FormBorderStyle**" to FixedToolWindow.**



6. Design the User interface elements on the form as below



The names of the controls are: 'btnFinish', 'txtName', 'txtX', 'txtY', 'txtZ'

Implementation

Our Modify Equiment Command will Inherit from BaseGraphicCommand.

We add EquipmentMiddle.DLL reference to our project as we deal with Equipment functionality, Add private references to objects we'll be using: the form, the Equipment, the Transaction Manager, and the UOM Manager. We also add a m_ModalState field to make command Modal if select set didn't contain ONE equipment before our command was started.

```
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle
Imports Ingr.SP3D.Common.Middle.Services
Imports Ingr.SP3D.Equipment.Middle
'import this to avoid 'long names' for mouse button
'constants in OnMouseMove and OnMouseDown
Imports Ingr.SP3D.Common.Client.Services.GraphicViewManager.GraphicViewEventArgs



Public Class EqpModCmd
    Inherits BaseGraphicCommand

    Private WithEvents m_frmEqpModForm As frmEqpMod
    Private m_oEqp As Equipment
    Private m_oTransactionMgr As ClientTransactionManager
    Private m_oUOM As UOMManager
    Private m_ModalState As Boolean
```

First let's implement the form:

In our form, we need to define the **Properties** it exposes to Command

Events that it **Raises** to Command – we basically want to be able to

1) get/set EqpPosition, Name on the Form
2) know when the user pressed the 'Finish' button, changed the name or changed the position.

```
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle.Services
Imports Ingr.SP3D.Common.Middle

Public Class frmEqpMod

    Public Event OnFinish()
    Public Event OnNameChanged()
    Public Event OnPositionChanged(ByVal oPos As Position)

    Private m_oUOM As UOMManager
    Private m_oPos As New Position
```

Implementation of Form

Our Form will have EqpPosition and EqpName properties (get/set). These will be used by the Command to set/get the position on the form. The form displays them when set by command on it.

We write a FormatDistanceValue function which we use in the set EqpPosition property logic to format and set the X,Y,Z fields for the specified position. This uses the FormatUnit method on UOMManager. We use Distance UnitType because Coordinates are of Distance type.

```
Public Property EqpPosition() As Position
    Get
        Return m_oPos
    End Get
    Set(ByVal value As Position)
        m_oPos.Set(value.X, value.Y, value.Z)
        txtX.Text = FormatDistanceValue(m_oPos.X)
        txtY.Text = FormatDistanceValue(m_oPos.Y)
        txtZ.Text = FormatDistanceValue(m_oPos.Z)
    End Set
End Property

Public Property EqpName() As String
    Get
        Return txtName.Text
    End Get
    Set(ByVal value As String)
        txtName.Text = value
    End Set
End Property
```

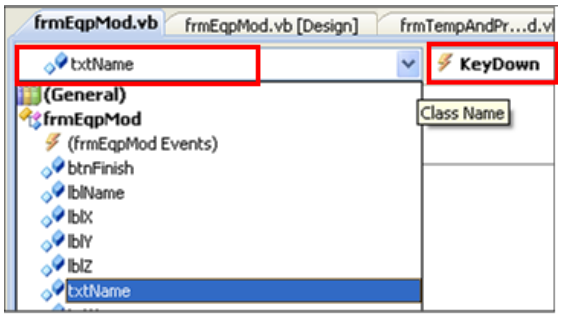
```
Private Function FormatDistanceValue(ByVal dVal As Double) As String
    Return m_oUOM.FormatUnit(UnitType.Distance, dVal)
End Function
```


Implementation of Form

We'll keep the business logic in our command and use the form to just gather data and raise events which the command logic reacts. Such separation of UI & business logic is a good practice, especially for ribbon bar like functionality. In the form designer, double click the **Finish** button – the event handler for it will be created and we add code – we simply raise the OnFinish() event.

```
Private Sub btnFinish_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles btnFinish.Click
    RaiseEvent OnFinish()
End Sub
```

We'll consider that the user changed name when 1) the Name text box loses focus or 2) 'Enter' key is pressed after typing the name. Using the drop-down combos at the top of the editor add the code for those 2 events to raise OnNameChanged():




```
Private Sub txtName_KeyDown(ByVal sender As Object,
    If e.KeyCode = Windows.Forms.Keys.Enter Then
        RaiseEvent OnNameChanged()
    End If
End Sub

Private Sub txtName_LostFocus(ByVal sender As Object
    RaiseEvent OnNameChanged()
End Sub
```

Implementation of Form …

We'll handle changes to the position coordinates similarly. We have to safeguard against invalid user input.

We know Coordinate position is a Distance value. So we must use UOM Service to interpret user's input and get the DBU values which we use to set on the object.

We'll write a function that checks input value is valid and change the text color to RED when we cannot interpret the keyin value as a distance value.

Such visual feedback of erroneous input is a good feature to provide.

```
'called whenever a coordinate value is changed.
'It will try to parse the value
'if it can't, it will show the value in 'red'
Private Function CheckCoordinate(ByVal sender As Object, ByRef dValue As Double)

    Dim oTextBox As System.Windows.Forms.TextBox = sender
    Dim bResult As Boolean = True

    Try
        dValue = m_oUOM.ParseUnit(UnitType.Distance, oTextBox.Text)
    Catch
        bResult = False
        oTextBox.ForeColor = Drawing.Color.Red
    End Try

    Return bResult
End Function
```


Implementation of Form …

We write a function which checks all coordinate keyins are valid before raising OnPositionChanged event.

```
Private Sub ComputePosition()
    'only raise event if all 3 coordinates are valid
    If (txtX.ForeColor = Color.Black) _
    AndAlso (txtY.ForeColor = Color.Black) _
    AndAlso (txtZ.ForeColor = Color.Black) Then

        RaiseEvent OnPositionChanged(m_oPos)

    End If
End Sub
```

The code for handling changes to the X (and similarly for Y & Z) coordinate becomes

```
'handle coordinate text boxes KeyDown(Enter) event
Private Sub txtX_KeyDown(ByVal sender As Object, ByVal e As Sys
    Dim oTextBox As System.Windows.Forms.TextBox = sender

    'first change color to black since we are entering a value
    oTextBox.ForeColor = Drawing.Color.Black

    If e.KeyCode = Windows.Forms.Keys.Enter Then
        Dim dValue As Double
        If CheckCoordinate(sender, dValue) Then
            m_oPos.X = dValue
            ComputePosition()
        End If
    End If
End Sub
```

```
'handle coordinate text boxes LostFocus event
Private Sub txtX_LostFocus(ByVal sender As Object, ByVal e As S
    Dim dValue As Double
    If CheckCoordinate(sender, dValue) Then
        m_oPos.X = dValue
        ComputePosition()
    End If
End Sub
```

Later in the class, we will learn a simple **.net** technique to reduce the amount of code by handling all these three text box handling functions into one, using the Handles keyword on the method and using 'sender' info to know which text box the event happened.

Implementation of Form …

In Form Load() we need to get a reference to the UOM Manager and set the X, Y, Z textbox foreground color to Black, since color is used to 'store' the results from the input validity checks.

```
Private Sub frmEqpMod_Load(ByVal sender As System.Obj
    'get UOM Manager
    m_oUOM = MiddleServiceProvider.UOMMgr
    'Init color of Textfields which we use for status
    txtX.ForeColor = Color.Black
    txtY.ForeColor = Color.Black
    txtZ.ForeColor = Color.Black
End Sub
```

We could have had another variable which maintained the last state of the input validity check. But, our current approach serves our purpose and more importantly, this was to introduce the technique of visually indicating the erroneous inputs from user.

Next, we implement OnStart : Initialize Services, get the Eqp item, setup & show our Form with data, set up to listen to events from form …

Implementation of Command … OnStart( )

In **OnStart**()
- Get selected equipment from Select Set (make sure one is selected) – if not we set our m_modalState variable used in Modal property implementation to return Modal=true, so that system ends us on OnStart's completion)

- Show form as Non-Modal (we expect Graphic events!!)
 - Remember we made the Form TopMost, so that it doesn't get lost behind when we click on graphic view.

- Initialize form fields with equipment name & position

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal arg

  With ClientServiceProvider.SelectSet
    'check if we have exactly one equipment in the SelectSet
    If Not (.SelectedObjects.Count = 1 AndAlso _
        (TypeOf .SelectedObjects(0) Is Equipment)) Then
      MsgBox("Select One Equipment before running this command")
      m_ModalState = True ' setting Modal=true here to
      Exit Sub ' Hint the System to stop us after OnStart returns
    End If

    'get selected equipment
    m_oEqp = CType(.SelectedObjects.Item(0), Equipment)
  End With

  'create and display form
  m_frmEqpModForm = New frmEqpModCmd
  m_frmEqpModForm.Show() 'display as Modeless

  'initialize form Properties
  m_frmEqpModForm.EqpPosition = m_oEqp.Origin
  m_frmEqpModForm.EqpName = m_oEqp.Name

End Sub
```

```
Public Overrides ReadOnly Property Modal() As Boolean
    Get
        Return m_ModalState
    End Get
End Property
```

Handle Mouse Move (Left+Drag) …

In **OnMouseMove()** if left mouse button is pressed (i.e. press left mouse and drag):

- Set Equipment Origin and call TransactionManager.Compute to make system evaluate the changes made
  - Note that, as a by-product of Compute, Display gets updated in rubberband-mode (Hilite).

- Update Form fields to indicate the 3D world position at the Mouse location in the view.

```
Protected Overrides Sub OnMouseMove(ByVal view As I
    If (e.Button = MouseButtons.Left) Then
        'Set the origin of the Eqp and Compute
        m_oEqp.Origin = position
        m_oTransactionMgr.Compute()

        'Set Position on the Form.
        m_frmEqpModForm.EqpPosition = m_oEqp.Origin
    End If
End Sub
```

- Note here, we get the Equipment's Origin after compute and use that to set on the Form. This way, we always get the correct position to show up on the form (this could be different from the graphics mouse position if any constraints exist on the Equipment limiting it from moving to the position we set.

Handle Mouse Down (Left/Right), Esc KeyDown …

In **OnMouseDown()** Implementation

- if left mouse button is pressed – same as OnMouseMove:
- if right mouse button is pressed – stop command

```
Protected Overrides Sub OnMouseDown(ByVal view As
    If (e.Button = MouseButtons.Right) Then
        StopCommand()
    Else
        If (e.Button = MouseButtons.Left) Then
            'same as MouseMove so just delegate
            OnMouseMove(view, e, position)
        End If
    End If
End Sub
```

OnKeyDown() - stop command on Escape key

```
Protected Overrides Sub OnKeyDown(ByVal e As Ingr

  If e.KeyValue = ConsoleKey.Escape Then
    StopCommand()
  End If
End Sub
```

Handle Form events in Command Class …

OnFinish() – Commit and stop command

```
Private Sub m_frmEqpModForm_OnFinish() Handles m_frmEqpModForm.OnFinish
    m_oTransactionMgr.Commit("Custom Equipment Modification Command")
    StopCommand()
End Sub
```

OnPositionChanged ( ) – update position and call Compute to show Changes in Display.

```
Private Sub m_frmEqpModForm_OnPositionChanged(…
    m_oEqp.Origin = oPos
    m_oTransactionMgr.Compute()
    m_frmEqpModForm.EqpPosition = m_oEqp.Origin
End Sub
```

OnNameChanged() – update name using SetUserDefinedName().
No need for Compute() since no graphical change.

Note : However, if it were a naming parent of another object, a Compute here would trigger the name rule on dependent named objects.

```
Private Sub m_frmEqpModForm_OnNameChanged() Handles m_frmEqpModForm.OnNameChanged
    If m_oEqp.Name <> m_frmEqpModForm.EqpName Then
        m_oEqp.SetUserDefinedName(m_frmEqpModForm.EqpName)
    End If
End Sub
```


Implementation of Command … OnStop( )

In Command's **OnStop( )**, we close our form and issue a Transaction Abort, so that we don't leave uncommitted changes.

```
Public Overrides Sub OnStop()
    'Take care of the situation that form is not initiated.
    'See OnStart implementation, in case one eqp wasnt selected
    'OnStart() shows a message and exits without shows form.
    If (m_frmEqpModForm IsNot Nothing) Then m_frmEqpModForm.Close()
    m_oTransactionMgr.Abort()
End Sub
```

Our command implementation is complete now, and you can Run/Debug the command.

Few Observations …

What would happen if the Equipment has a Foundation Block attached… Foundation block is an independent object, not integral with Equipment, but is associated by port connections to the Equipment (Equipment Foundation port)

Similarly, if the Equipment you are moving has a mate/align relationship to another equipment, do you see any constraints during move attempt?

If the Equipment had a Nozzle which in-turn has connected “unconstrained and free to move” piping (say just a Flange and Pipe) , what happens to the Piping when the Equipment moves?

What would happen if we omit the Compute call in the OnMouseMove( ) ?

What would happen if we omit the Compute call in the OnMouseMove, and then hit finish?

Implementing multiple events with one Function

In this Form, we had a situation where we had to deal with X,Y,Z fields which had similar logic in their KeyDown and LostFocus methods. We already implemented txtX_KeyDown, txtY_KeyDown, txtZ_KeyDown (and similarly LostFocus) methods.

.net provides a better way to handle such events from multiple objects as below.

The key syntax points to note in this are

- The use of “**Handles**” keyword with more than one object events.
- The way we are able to find out which textbox this event occurred (“**sender**”).
- The use of “**Is**” operator to identify if sender is txtX / txtY / txtZ.

```
'handle coordinate text boxes KeyDown(Enter) event
Private Sub txtXYZ_KeyDown(ByVal sender As Object, _
                           ByVal e As KeyEventArgs) _
    Handles txtX.KeyDown, txtY.KeyDown, txtZ.KeyDown

    Dim oTextBox As TextBox = sender

    'change color to black as we are entering a value
    oTextBox.ForeColor = Color.Black

    If e.KeyCode = Windows.Forms.Keys.Enter Then
        Dim dValue As Double
        If CheckCoordinate(sender, dValue) Then
            If (sender Is txtX) Then m_oPos.X = dValue
            If (sender Is txtY) Then m_oPos.Y = dValue
            If (sender Is txtZ) Then m_oPos.Z = dValue
            ComputePosition()
        End If
    End If
End Sub
```

Implementing multiple events with one Function

Similarly the LostFocus methods of txtX, txtY, txtZ controls can be combined into one as below.

```
'handle coordinate text boxes LostFocus event
Private Sub txtXYZ_LostFocus(ByVal sender As Object, _
                             ByVal e As System.EventArgs) _
Handles txtX.LostFocus, txtY.LostFocus, txtZ.LostFocus
    Dim dValue As Double
    If CheckCoordinate(sender, dValue) Then
        If (sender Is txtX) Then m_oPos.X = dValue
        If (sender Is txtY) Then m_oPos.Y = dValue
        If (sender Is txtZ) Then m_oPos.Z = dValue
        ComputePosition()
    End If
End Sub
```


Implementing multiple events with one Function …

Even simpler, one can merge both KeyDown and LostFocus handling into just one function as below. This alone suffices instead of the previous two.

```
' Even Simpler - Just ONE routine handling both LostFocus and KeyDown
Private Sub txtXYZ_LostFocusOrKeyDownEvents(ByVal sender As Object, _
          ByVal e As System.EventArgs) _
        Handles _
          txtX.KeyDown, txtY.KeyDown, txtZ.KeyDown, _
          txtX.LostFocus, txtY.LostFocus, txtZ.LostFocus

  If (TypeOf e Is KeyEventArgs) Then
    If CType(e, KeyEventArgs).KeyCode <> Keys.Enter Then Exit Sub
  End If

  Dim val As Double
  If CheckCoordinate(sender, val) Then
    If (sender Is txtX) Then m_Pos.X = val
    If (sender Is txtY) Then m_Pos.Y = val
    If (sender Is txtZ) Then m_Pos.Z = val
    ComputePosition()
  End If
End Sub
```

Further Enhancements … Use RibbonBar instead of Form

We will write an enhanced command (EqpModRibbonBarCmd) to use a RibbonBar instead of a Form.

To create a RibbonBar,

- Just create a New User Control in the project (named EqpModCmdRB)
- Inherit it from *BaseRibbonBarControl* instead of *UserControl* (edit the RibbonBar's designer.vb file),
- Add your controls, expose properties / events to users (Commands)

Our RibbonBar uses the same controls/logic as form did, so we can re-use that UI.

- Select Controls on Form, Copy, and Paste on the RibbonBar,
- The Events, Properties we used to link up the Form & Command are going to be the same for RibbonBar as well. So we copy / paste all that code as well.

To set the RibbonBar of the command, in Command's OnStart ( ),

- we create a RibbonBar instance, assign it as Command's RibbonBar property.
- Initialize RibbonBar fields – i.e. Set properties on RibbonBar.
- Note that no OnResume/Suspend handling is needed.
- In OnStop ( ) we set the Command's RibbonBar property to Nothing, and Abort.

S3D .net API

Math

–

Position, Vector, Matrix4x4 Transformations

Position

- Use **Position** Class to create a 3D position.
- Offers required constructors, properties, methods and operators.

- Can construct a Position with Coordinates, another position.
- **X, Y, Z** properties → coordinates of this Position.

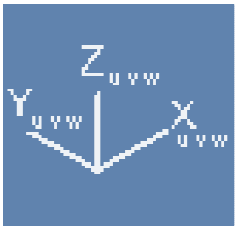
- **Get** (x,y,z) / **Set** (x,y,z)→ gets / sets x,y,z coordinates of this position
- **DistanceToPoint** (*position*) → gets distance to another *position*.
- **Offset** (*vector*) → moves the Point by given *vector* (distance & direction).
- **Subtract** (*position2*), as well as **–** operator → returns the vector from *position2 to* this *position*.
- '*****' operator transforms a given *position* with a given Transformation *Matrix4x4*, and returns the transformed position.

Vector

- Use **Vector** Class to create a 3D Vector.
- Offers required constructors, properties, methods and operators.
- Can construct a Vector with x,y,z, components or another vector.
- **X, Y, Z** properties → components of this Vector.
- **Length** property → Vector's magnitude. Setting Length scales the vector. Length = 1 makes it unit vector.
- **Get** (x,y,z) / **Set** (x,y,z) → get / set x,y,z components of this vector
- **Dot**(*vector*) → returns dot product (magnitude) with given vector.
- **Cross**(*vector*) → returns vector representing Cross product with given vector.
- **Add**(*vector*) as well as '+' operator → Adds two vectors.
- **Subtract**(*vector*) as well as '–' operator → Subtracts two vectors.
- '*' operator returns vector representing cross product of two vectors.
- '*' operator transforms this *vector* with given *Matrix4x4* and returns the transformed vector.
- **Angle** (*vector, normal*) → calculates angle (in radians) with given *vector.* The direction is determined by the *normal* using the right-hand rule.

Matrix4x4

- Use **Matrix4x4** Class to create a 3D transformation matrix.
- Look at it as a Coordinate system with origin & x,y,z vectors.
- Offers required constructors, properties, methods and operators.
- **Get** (values( )) / **Set** (values( )) → methods to Get/Set matrix values into/from a double array.
- **GetIndexValue(***index***)** / **SetIndexValue** (*index, value*)→ get /set *value* at *index*.
- **SetIdentity** ( ) → method to initialize the matrix to a *identity* matrix.
- **Rotate** (*angle, vector*) → rotates the matrix by given *angle* (radians) about given *vector*.
- **Translate** (*vector*) → moves the origin (x_{pos}, y_{pos}, z_{pos}) of the matrix by the given *vector*.
- **Invert ( )** → Inverts the matrix.
- **MultiplyMatrix**(*matrix4x4*) → multiplies this *matrix* with given *matrix4x4*.
- **Transform** (*position*) → returns matrix transform of a given *position*.
- **Transform** (*vector*) → returns matrix transform of a given *vector*.



x_{ux}	y_{uy}	z_{uz}	x_{pos}
x_{vx}	y_{vy}	z_{vz}	y_{pos}
x_{wx}	y_{wy}	z_{wz}	z_{pos}
x_{scale}	y_{scale}	z_{scale}	scale

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

IndexValues

Transformations

To move a ***position*** along a ***vector*** by a given ***distance***, create the ***vector***, set its *Length* as the ***distance*** to move, and then use the Position.Offset (vector) method call which returns the moved Position.

```
Dim oPos as Position, oVec as Vector, oMovedPos as Position ' Assume oPos, oVec
    are initialized after this before the below statements are executed.
oVec.Length = distance
oMovedPos = oPos.Offset (oVec)
```

To arrive at the **transformation matrix** for a **move operation**, ***initialize*** a new Matrix4x4 to Identity Matrix and then ***Translate*** it by a given vector which represents the delta distance to move in x, y, z directions (in metres).

```
Dim oTransVec As New Vector, oXformMtx as New Matrix4x4
oTransVec.Set deltaX, deltaY, deltaZ
oXformMtx.SetIdentity
oXformMtx.Translate oTransVec
```


Transformations …

To arrive at a **Transformation matrix** for a **rotate operation** about a RotationAxis by an angle w.r.t a RotationPoint, you would do the below steps.

```
Dim oRotAxis As New Vector, oRotPt as New Position, oRotMtx As New Matrix4x4
Dim oXlateMtx As New Matrix4x4, oInvXlateMtx As New Matrix4x4, oXformMtx
    As New Matrix4x4
' assumes oXlateMtx , oInvXlateMtx, oXformMtx, oRotMtx are initialized to
    Identity
' initialize translation matrix and its inverse, assumes oRotPt is initialized already
oXlateMtx.Translate (New Vector (–oRotPt.x, –oRotPt.y, –oRotPt.z) )
oInvXlateMtx.Translate (New Vector (oRotPt.x, oRotPt.y, oRotPt.z) )
oRotMtx.Rotate dAngle, oRotAxis ' Finalize Rotation Matrix for dAngle rotation
    about oRotAxis.
oXformMtx.MultiplyMatrix oInvXlateMtx ' Nullify Translation Portion
oXformMtx.MultiplyMatrix oRotMtx ' Rotate it
oXformMtx.MultiplyMatrix oXlateMtx ' un-nullify Translation Portion.
    Transformation Matrix finalized.
```

Transformations …

To get the **Local Coordinate System Position** from **Global Coordinate System Position**, do the following.

```
Dim oGlobalPos as Position, oLocalPos as Position, oLCSMtx as Matrix4x4,
    oLCSInvMtx as Matrix4x4
oLCSInvMtx = New (oLCSMtx) ‘ Assumes oLCSMtx is initialized by now.
oLCSInvMtx.Invert ‘ get Inverse of LCS.
oLocalPos = oLCSInvMtx.Transform (oGlobalPos)
```

Transforming 3D Objects

On any S3D Object implementing **ITransform**, you can call the Transform ( ) method [available on the ITransform interface] passing the Transformation matrix which you want to apply as parameter. This will do the necessary Transformation. This is how one can do a generic Move/Rotate/Mirror transformation of an S3D object.

S3D .net API

LAB

–

Math – Position, Vector, Transformation Matrix, Coordinate System Transforms

LAB - Overview

- In this LAB we will learn some Math Functionality usage
  - Position, Vectors, Matrices
  - Coordinate Transformations
- We will be enhancing the Graphic command we wrote earlier in VB.net which displayed the View and World coordinates of the mouse pointer in a form. The enhancements would be
  - If **First** element in the SelectSet is **Equipment**, then Get its Matrix. Otherwise, uses a Unit Matrix.
  - Treating that Matrix's Coordinate System as a Local Coordinate System (LCS), get the LCS coordinates of the World Coordinates at the current Mouse Position in view.
  - Get x,y,z Components of Vector [in the **Global** Coordinate System] between the mouse Position World Point and Position of the Matrix's origin. These represent distances from the Current Point to the Equipment in GCS.
  - Report the LCS coordinates and the vector X, Y, Z values on the Form in addition to the Global Coordinates currently reported.

Add references, Imports directives, class variables

- Since we need to get the Equipment's matrix, we add the EquipmentMiddle.dll reference to the project
  - (Right Click on References, Add Reference … > Browse > and choose $Core\Container\Bin\Assemblies\Release\EquipmentMiddle.dll)
- Since we need to refer to Equipment in our code, we add the below imports directive in the top section of the GraphicCmdLab.vb

```
Imports Ingr.SP3D.Equipment.Middle
```

- If the select set contains an Equipment, we will arrive at a transformation matrix based on the Equipment's matrix. We also need its Position. We will use these in MouseMove event handler to do a coordinate transform and vector math. Hence, we add them as class private member variables.

```
Private m_frmCmdForm As frmGraphicCmdLab
Private m_oMtx As Matrix4X4
Private m_oOrigin As New Position ' Inits to 0,0,0
```


Arrive at Global CS → Local CS Transformation Matrix.

In OnStart( ) we check if first object in SelectSet is Equipment. If so, we instantiate our ***m_oMtx*** member variable by constructing a Matrix4x4 passing the Eqp's Matrix as parameter(using *copy* constructor). We then invert the matrix, as we learnt in our Coordinate Transformations topic in Math Session.

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argumer
    With ClientServiceProvider.SelectSet.SelectedObjects
        If (.Count >= 1) AndAlso (TypeOf .Item(0) Is Equipment) Then
            ' Get the Equipment
            Dim oEqp As Equipment = .Item(0)
            ' Create a New Matrix from Eqp's Matrix & Get its Origin.
            m_oMtx = New Matrix4X4(oEqp.Matrix)
            m_oOrigin.Set(m_oMtx.GetIndexValue(12), _
                          m_oMtx.GetIndexValue(13), _
                          m_oMtx.GetIndexValue(14))
            m_oMtx.Invert() ' Invert the Matrix.
        Else
            m_oMtx = New Matrix4X4 'No Eqp, create new Matrix
        End If
    End With

    m_frmCmdForm = New frmGraphicCmdLab
    m_frmCmdForm.Show()
End Sub
```

MouseMove : Transform MousePoint to LCS, Get Vectors

In OnMouseMove( ) we transform the mouse Position using the Transformation matrix to get the Local Position and then report both the Global Position and Local Position. We then get the vectors from the Origin of the Matrix to the Mouse Point and show.

```
Protected Overrides Sub OnMouseMove(ByVal view As Ingr.SP3D.Common.Client.Services.Grap

    With m_frmCmdForm
        .lblViewXValue.Text = e.X
        .lblViewYValue.Text = e.Y

        Dim oLPos As Position = m_oMtx.Transform(position) ' Transform GCS -> LCS
        Dim oVec As Vector = (position - m_oOrigin) 'Get Vector from position to origin
        .lblWorldXValue.Text = "Gx:" & position.X & " Lx:" & oLPos.X & " Vx:" & oVec.X
        .lblWorldYValue.Text = "Gy:" & position.Y & " Ly:" & oLPos.Y & " Vy:" & oVec.Y
        .lblWorldZValue.Text = "Gz:" & position.Z & " Lz:" & oLPos.Z & " Vz:" & oVec.Z
    End With
End Sub
```

Our command implementation changes are complete. You can Run/Debug the command. At runtime, just widen the form to see the full text.

Improvement – Get Intelligent (snapped) points

Instead of just using 3D Position from View, we can get an intelligent (snapped) point from SmartSketch, using the following approach …

1. Define a class variable of **SmartSketch3d** type → `Private m_oSmartSketch As SmartSketch3d`

2. Initialize (**new**) it in **OnStart** → `m_oSmartSketch = New SmartSketch3d`

3. Call **Smartsketch3d.GetPosition** passing in the mouse event details (View, X, Y, event). It solves with active Options and returns snap point or usual 3D Position if none. It also hilites and returns the BusinessObject influencing the solved point, if any.
(You can also get all the Solved Constraints by calling GetSolvedConstraints method)

```
Protected Overrides Sub OnMouseMove(ByVal view As Ingr.SP3D.Common.Client.Services.GraphicView, ByVal e As Ingr.SP3D.Commo
  Dim oLCSPt As Position = m_oMtx.Transform(position)
  Dim oVec As Vector = (position - m_oOrigin)
  Dim oSS3DPt As Position = Nothing, oBO As BusinessObject = Nothing

  m_frmCmdForm.lblViewX.Text = e.X
  m_frmCmdForm.lblViewY.Text = e.Y
  m_oSmartSketch.GetPosition(view, e.X, e.Y, SmartSketch3dMouseEvent.Move, oSS3DPt, oBO)
  m_frmCmdForm.lblWorldX.Text = "GCS X:" & position.X & " SS3D X:" & oSS3DPt.X & " LCS X:" & oLCSPt.X & " Vec X:" & oVec.X
  m_frmCmdForm.lblWorldY.Text = "GCS Y:" & position.Y & " SS3D Y:" & oSS3DPt.Y & " LCS Y:" & oLCSPt.Y & " Vec Y:" & oVec.Y
  m_frmCmdForm.lblWorldZ.Text = "GCS Z:" & position.Z & " SS3D Z:" & oSS3DPt.Z & " LCS Z:" & oLCSPt.Z & " Vec Z:" & oVec.Z
End Sub
```

Our command improvements are complete now, and you can Run/Debug the command.

S3D .net API

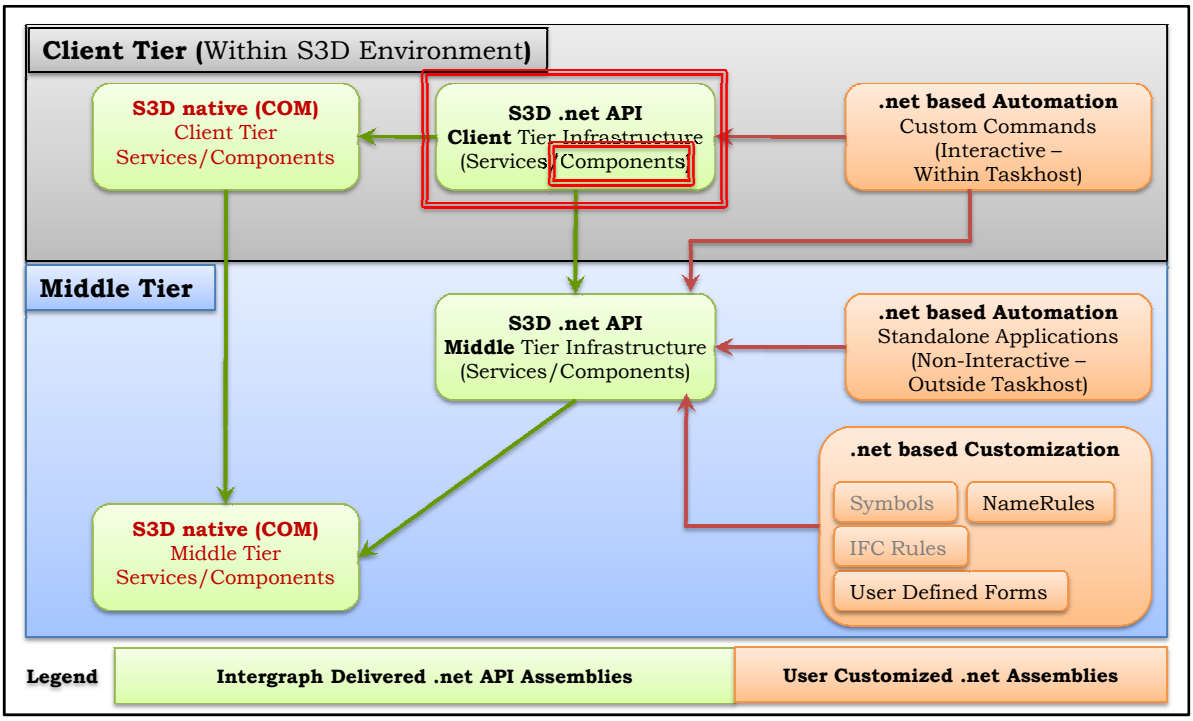
Client Components

-

Automation Components

Related to User Interface

S3D .net API – Client Services



Client Components

- Some special client components are essential to write Interactive commands – for example .
  - Interactive Commands often have a need to ***Hilite*** objects of interest. For example, a "Show Free Ends" command on a Pipeline would do the processing to find free ends and then add them to a ***Hiliter***.
  - Interactive Commands often provide user with a facility to draw a ***Rectangular Fence*** by two points to give a visual indication of a portion of the view, probably for locating some kind of objects within the region.
  - Interactive commands sometimes want to ***Locate*** objects at a Point, or within a rectangular region (often done with a rectangular Fence)
  - Interactive Commands often want to provide ***SmartSketch*** assistance to user in selecting intelligent points – snapping to Object Ends, Center, PointOnLine, Surface, DivisorPoints, Directional constraints w.r.t. a given position etc.

Hiliter / GraphicViewHiliter

- Just create a new **GraphicViewHiliter** / **Hiliter** and use.
- *GraphicViewMgr,* available from ClientServiceProvider, also provides CreateHiliter ( ) method. [do not use – being deprecated]
- ***GraphicViewHiliter*** → Both persistent/transient objects – graphic views only.
- ***Hiliter*** → persistent objects only – all views (WSE, TODO List, GraphicViews, PID etc)

Hiliter Functionality

- Temporary Display Hilite.
- Any number of Hiliters can be created (2-3 enough at a time for practical situations).
- Specify Color, LinePattern, Weight, Clipping behaviour (Hilited graphics clipped/not).
- Maintains Hilited Objects Collection. Can
  - Add / Remove Object(s) to/from Hilite List.
  - Check if Hilited List Contains a given Object.
- Hilited Objects List cleared on Transaction.Commit / Abort.

GraphicView, RectangularFence, Locator

- *GraphicViewMgr,* available from ClientServiceProvider, provides ActiveView property which is the ***Active*** GraphicView.
- Graphic & StepCommand Base classes also provide CurrentView & PreviousView properties, both are of type GraphicView.

GraphicView Functionality

Create Locator

Create RectangularFence

Get/Set MousePointer

RectangularFence Functionality

Start/Drop Fence

Set Initial & Moving Points

Get Top-Left and Bottom-Right Points

Hide/Show Fence

Control Line Color, Width, Style, Stipple Factor

Locator Functionality

Locate Objects (in its GraphicView)

- Around a Point within specified pixel Radius
- In a given Rectangular Region, with and without overlap options
- Within an Object, eg Sub-Geometry within an Equipment.

Control Locate Depth Limit

Can specify List of Objects to be Excluded from Locate.

SmartSketch3D [SS3D]

- ***SmartSketch3D,*** a newable object, provides "point solving" Functionality.

Enabled [get/set]	whether or not SS3D is enabled.
Projection [get/set]	Projection option – (whether/not to project point to a plane).
ProjectionMatrix [get/set]	The projection plane matrix.
AssistingPosition [get/set]	Position to solve offset/parallel/perpendicular/Align constraints.
List [get/set]	SS3D's global LocateList. (used for constraint solving)
Constraints [get]	The Constraints used to solve the returned position.
Options	SS3D settings. i.e. [get/set] various point solving options.
GetPosition ( )	"solved 3D position", its driving object(s) if any, for the mouse {x,y} in graphic view, solved with the specified constraints.
SolvedConstraints [get]	The solved constraints evaluated with GetPosition.
AddExcludeFromLocate ( )	Specify object(s) to exclude from Locate.
ClearExcludeFromLocate ( )	Clears the Locate exclusion list set if any.
Tolerance, Depth, Flags, Target [get/set]	Locator Parameters (Tolerance, Pixel depth, Locator Flags, Target to Locate into)
LocatedObjects [get]	Objects located by SS3D.
Filter [get/set]	object Implementing Filter to use by SS3D.
Hiliter [get]	the Hiliter used by SS3D.
Events From SS3D	ObjectsLocated, Projection / Position Changed, ConstraintConflict

- Can use in a Command/Assistant. A Step Command can also use SS3D inherent in the Step if it's Locate Behavior is set to SmartSketch3D.

SmartSketch3DList

- SmartSketch3D internally maintains a list of Business Objects, which it automanages based on user mouse hover actions over objects.
- A command can also manage it according to its needs.
  - (refer to interactive “Add to SmartSketch list” command.)
- ***SmartSketch3DList*** is the class to manage the same using code. This can be done by a command if it needs to do so.
- Just new it, finalize its objects list using the methods explained below and set it on the ***SmartSketch3d*** object’s ***List*** property.

Add ( )	Add an Object to List
Remove ( )	Remove an Object from List
Clear ( )	Clears the List.
Count	Gets Count of Objects
Maximum	get/set the list size default = 5, max value = 10, min value = 1
Contains (object)	Checks if an object exists in the list.
Item (object)	access item by index/key

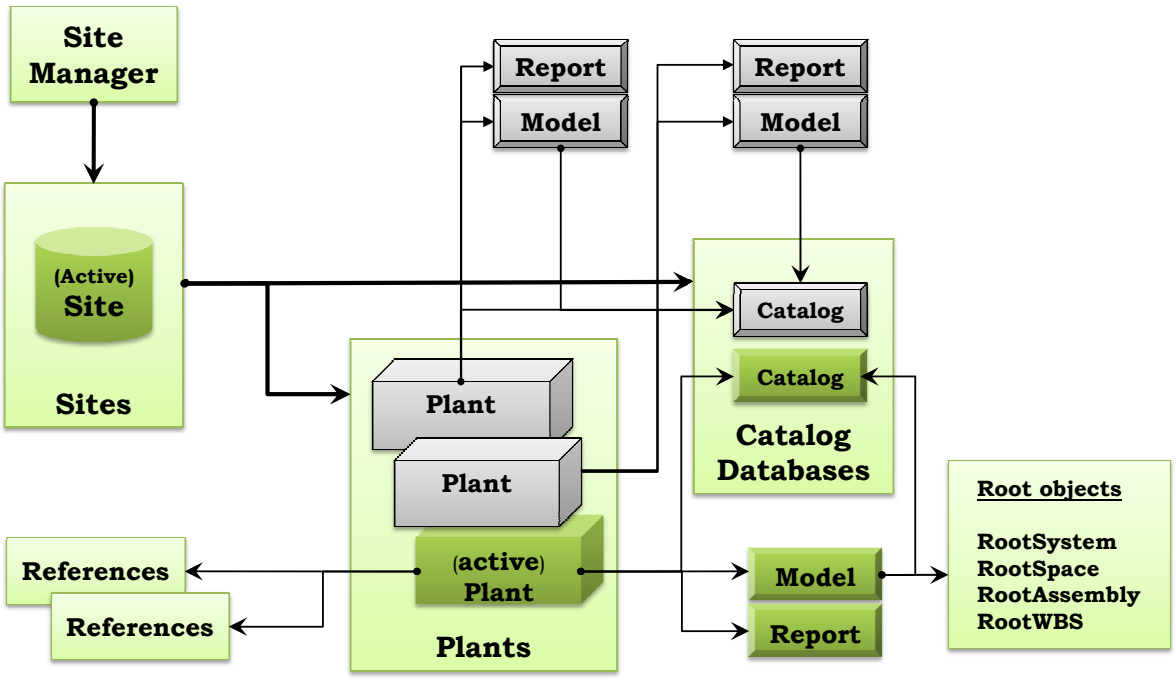
S3D .net API

Accessing

Site, Plant, & Plant Connections

Model, Catalog, Reports, References

Site, Plants, Model, Catalog, References



SiteManager, Site

Site Manager

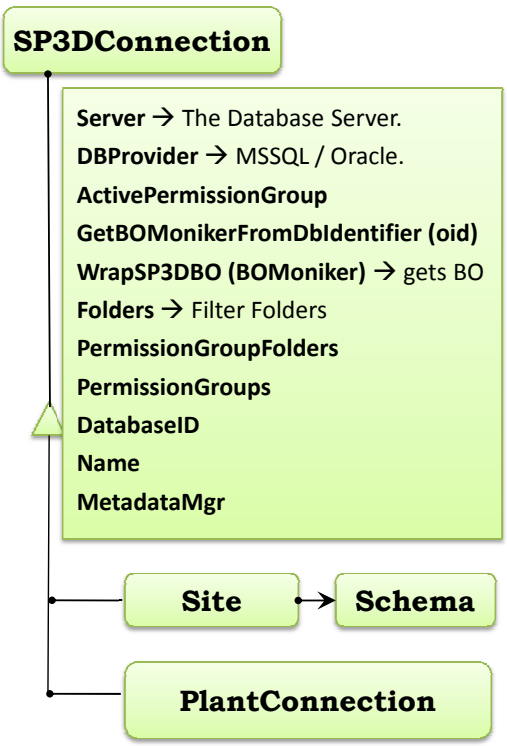
- Sites() → Collection of Sites.
  - Today S3D knows one Site, the one configured using "Modify Database and Schema Location" utility.
  - This property is provided as collection to enable future access to all available sites…
- ConnectSite ( ) → Method to connect to a Site. Useful in Standalone Apps. Cannot use in Interactive Commands as active Site is already is activated when inside S3D taskhost.
- ActiveSite property
- SiteServerName property

Site

- Name → Name of the Site
- Plants() → Collection of Plants.
- OpenPlant ( ) → Method to open a Plant by given name, Useful in Standalone Apps. Otherwise, within S3D taskhost, a ActivePlant already set.
- ActivePlant Property → Currently Active Plant.
  - Cannot change once set
- CatalogDatabaseInfos( ) → Information of all Catalog Databases in the Site, Their SymbolShare Locations etc.

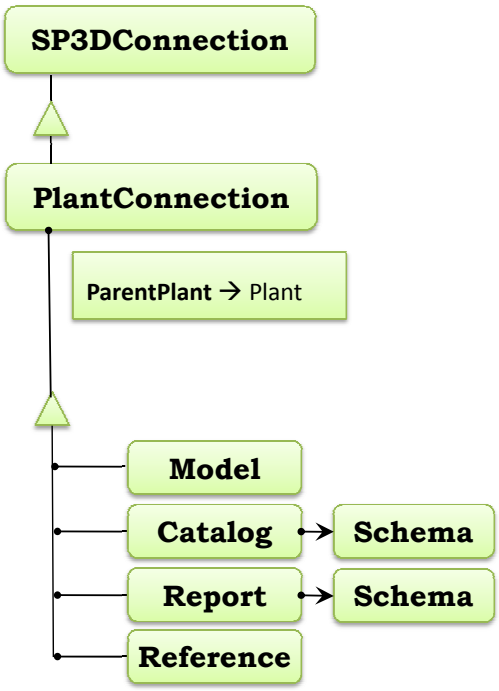
Site, SP3DConnection and Plant Connections

- A Site and **Plant** connections are in a way Data Connections i.e.
  - **Site & Site Schema**
  - **Model**
  - **Catalog** & Catalog **Schema**
  - **Reports** & Reports **Schema**
  - **References**
- **SP3DConnection** is the base class of them all, and provides common functionality, aptly overridden by derived classes when required.



Plant Data Sources / Connections

- A **Plant** has several Data Sources / Connections i.e.
  - **Model**
  - **Catalog** & Catalog **Schema**
  - **Reports** & Reports **Schema**
  - **References**
- **PlantConnection** is the base class of them all, and provides common functionality, aptly overridden by derived classes when required.



Plant, Model

Plant

- **Name** → Name of the Plant
- **Description** → Description of the Plant
- **PlantModel** → Model Connection
- **PlantCatalog** → Catalog Connection
- **PlantReport** → Reports Connection
- **References ( )** → Reference Connections
- **AddReference( )** → To add new DGN file/DWG file/PDS project.

Model

The Model Connection of the Plant.

Provides access to below information/objects.

- **PlantCatalog** → Catalog Connection
- The root objects of the plant, i.e.
  - **RootSystem**
  - **RootAssembly**
  - **RootSpace**
  - **RootWBS**

Catalog, Reports, Schema Info, References

Catalog

The Catalog Connection of the Plant. Provides access to below information/objects.

- **PlantModels** → Models referring this Catalog
- Access to objects and information in the Catalog is via **CatalogHelpers**, which we discuss later.

Catalog Schema & Reports Schema

The Schema information is available from the MetadataMgr of the respective connections, i.e.

Model, Catalog, Reports, References

Reports

The Reports Connection of the Plant. Provides access to below information/objects.

- **PlantCatalog** → The Catalog of the associated Model of this Reports Database.

References

The Reference Connections of the Plant. Provides access to below information.

- **ReferenceType** → The Type of the Reference, i.e.
  - PDS
  - MicroStation
  - Autocad

S3D .net API

Accessing 3D Schema Information

Metadata Manager

What is Metadata

- In short – **metadata** is "**information**" about "**data**" - objects, classes, interfaces, properties …
- Before any object can be dealt with, or any property can be referred to, or any interface can be used, or any relationship can be established, it must be well described in the system as to
  - What Type it is (Type of Class, Type of Property …)
  - What is its Name and Displayed Name (of Class / Interface / Property / Relationship Pattern / Category)
  - Its internal unique ID assigned for that type (called the GUID or IID or DispID for classes, Interfaces and Properties respectively)
  - Category of the Property, whether it is shown on Property Page or not, whether it is user modifiable or not, what is the property's data type (Long, Double, Codelisted, String, …), whether unitted property or not, if unitted property, what are the Units in which the property value is interpreted in…
  - In case of Interfaces, Systemic behavior as to what all semantics are associated with an interface, i.e. what internal solvers run when a property belonging to that interface is modified…

What is Metadata…

- System propagation behavior as to what happens on delete/copy/paste/modify etc on a class/interface…
- The roles associated in a Relationship definition, what is the cardinality of the involved objects in the relationship pattern, what are the involved interfaces in the relationship pattern…
- What Business Object Classification does a given object's class belong to…
- What Business Object Classification has what common interfaces and defining interfaces…
- What codelist values exist in a given codelist table…
- What are the Parent & Child codelist tables of a given hierarchical codelist table…
- Such Metadata is the secret to success of the Smartplant/SmartMarine 3D Solution. The system relies on Metadata to operate on different kinds of objects, their inter-relationships and auto-reacts to changes / modifications, reduce a number of actions to do by user, eliminate obscure steps, making it intuitive, productive and workflow focused.

MetadataManager

- Provides Metadata information
  - Classes, Interfaces, Properties, Codelists, Relationships, BOC, Edges
- Available as
  - collections (ReadOnlyDictionary (Of <*Type*>Information)
  - Methods to get individual item: Get<*Type*>Info
- eg,
  - ReadOnlyCollection<ClassInformation> **Classes**
  - ClassInformation **GetClassInfo**(string className)
  - ReadOnlyCollection<InterfaceInformation> **Interfaces**
  - InterfaceInformation **GetInterfaceInfo**(string interfaceName, string nameSpace)
  - ReadOnlyCollection<CodelistInformation> **Codelists**
  - InterfaceInformation **GetCodelistInfo**(string codelistName)

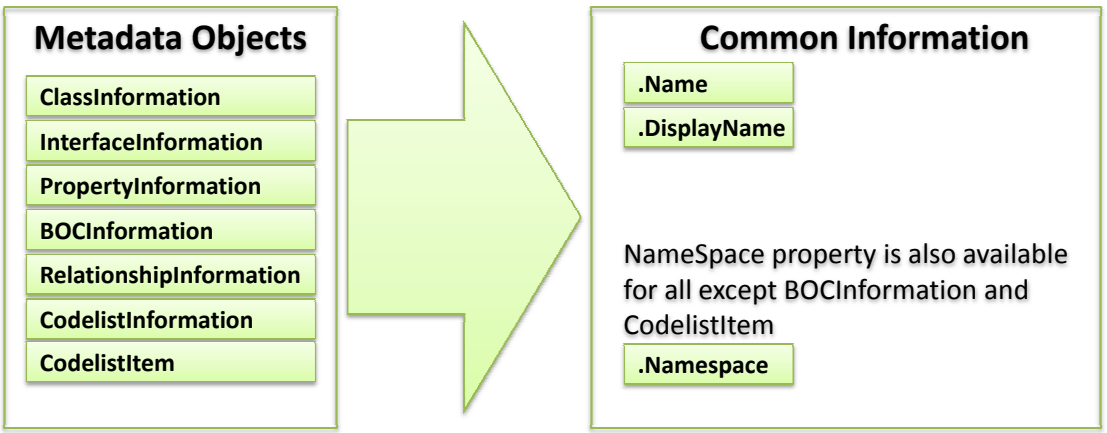
MetadataManager …

- Access from Model, Catalog, Report and References.
  Like ***Model.MetadataMgr***
- Provides properties & methods to access metadata – Classes, Interfaces, Relationships, BOCs, Codelists, Edges
- Prefer individual access methods rather than accessing entire collections

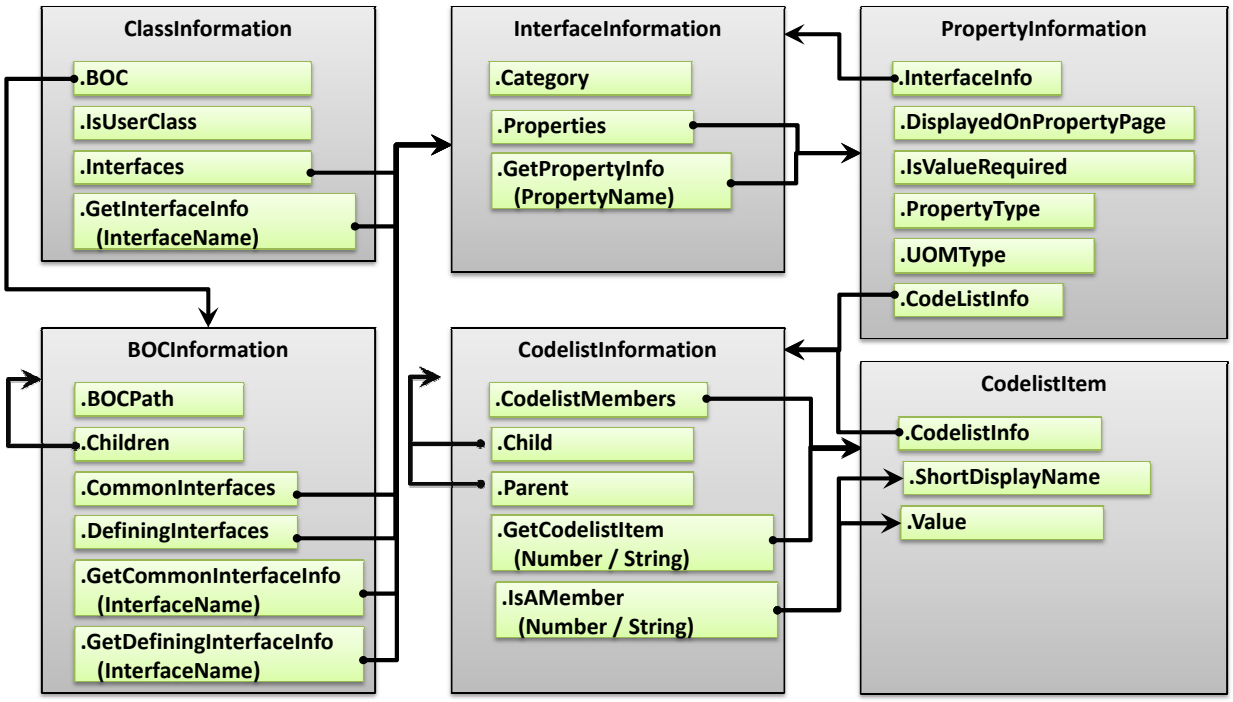
Querying Entire Metadata	Querying Specific Metadata (preferred)
.Classes	.GetClassInfo (ClassName)
.Interfaces	.GetInterfaceInfo (InterfaceName)
.Relationships	.GetRelationshipInfo (RelationshipName)
.BOCs	.GetBOCInfoByPath (BOCPath)
.Codelists	.GetCodelistInfo (CodelistName)
.Edges	.GetEdgeInfo (EdgeName)

Common Information on Metadata Objects

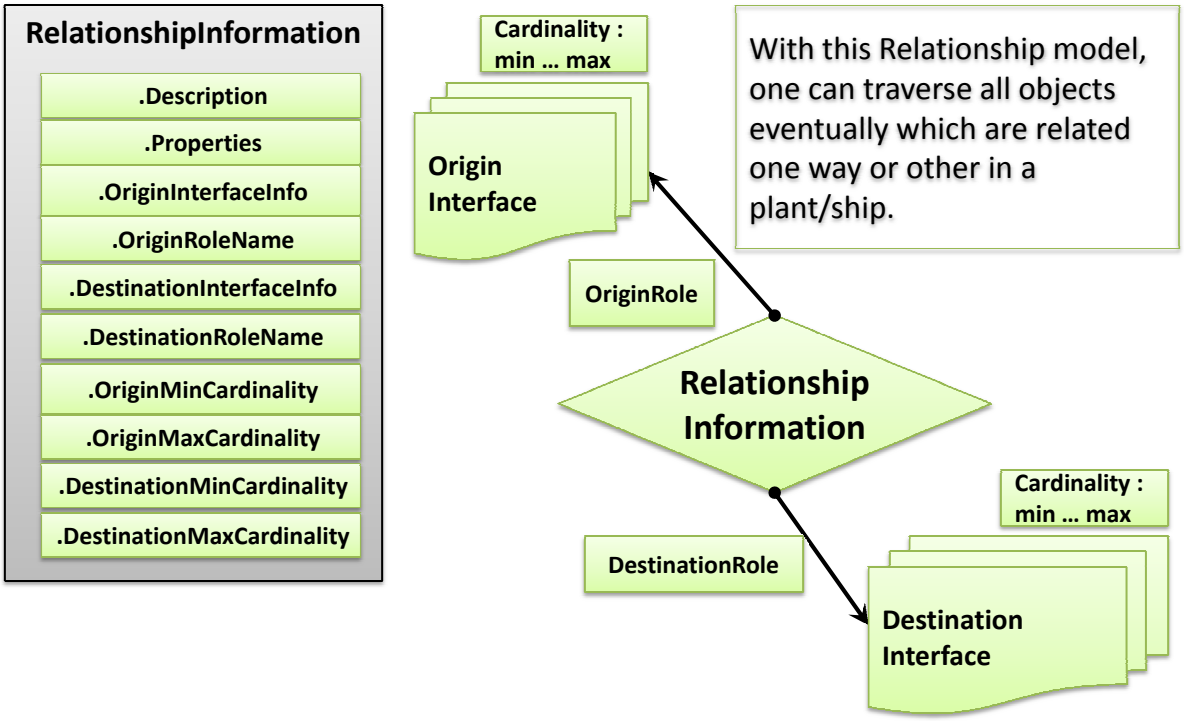
- Below Information is common to all Metadata Objects.



Class, Interface, BOC, Property, Codelist Info…



Relationship Information



S3D .net API

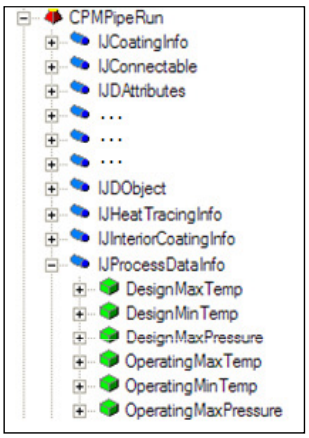
Generic Property Access

-

Get/Set properties

Generic Property Access – Get

- **BusinessObject**, the base class of all the SmartPlant objects, provides ***Generic Property Access*** mechanism to access any property on it.
- The word “Property” and “Attribute” are used interchangeably and mean the same.
- The property must be defined and COM & SQL visible in Metadata to be accessible in this manner.
- Works for User Defined attributes as well. (bulkloaded via CustomInterfaces & CustomClassInterfacesList sheets)
- To access an attribute, you need to know its InterfaceName and AttributeName. MetadataBrowser helps to find them.
- Property value could be null for nullable attributes



Generic Property Access – Get

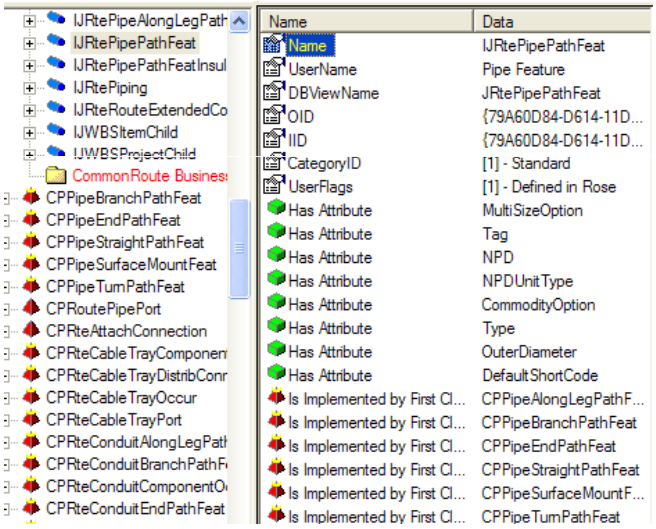
For people who know SP3D COM object model, this is similar to using IJDAttributes to access and modify attributes.

GetPropertyValue is the method to use, and there are two overloads of it as below.

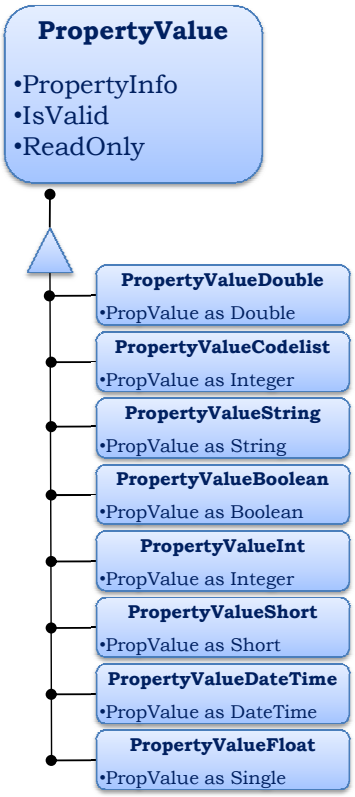
GetPropertyValue (interfaceName, attributeName) As **PropertyValue**

GetPropertyValue (PropertyInfo) As **PropertyValue**

PropertyValue Class is the BaseClass of different PropertyValueXXXX classes supported by S3D, i.e. Boolean, Double(8bytes), Float(4bytes), Integer(4bytes), Short(2bytes), String, Codelist and DateTime



Property Value Classes



All of the **PropertyValueXXX** classes have constructors to create respective **PropertyValueXXXX** type objects with below inputs

- interfaceName, attributeName, value
- PropertyInfo, value

PropertyValueCodelist also has a constructor with below inputs

- PropertyInfo, CodelistItem

The Boolean, DateTime, Short, Int, Float, Double types have means to specify a Null value, using the nullable variable declaration syntax.

```
Dim iValue As Nullable(Of Integer) ' or Dim iValue as Integer?
  iValue = oPropValueInt.PropValue
  If (iValue is Nothing) Then
     …
  End If
```

PropertyValueXXX classes allow on to predefine a PropertyValueXXX object and apply such property to each BusinessObject in a BusinessObject collection, like below.

```
Dim oBOs = oFilter.Apply( ) ' assume filter set.
Dim oPV as New PropertyValueDouble("IJUAVDims", "Length", 5.0)
For Each oBO in oBOs
    oBO.SetPropertyValue(oPV);
Next
```

Generic Property Access – Set

- To modify an attribute, Use **SetPropertyValue**. Several overloads and datatype variants exist.
  - SetPropertyValue (value, interfaceName, attributeName) ' → for each data type
  - SetPropertyValue (value, PropertyInfo) ' → for each data type
  - SetPropertyValue (PropertyValue) ' Notice the use of PropertyValue BaseClass, which caters for all PropertyValueXXX classes.

- For Codelisted properties can use below variants as well.
  - SetPropertyValue (CodelistItem, interfaceName,attributeName)
  - SetPropertyValue (CodelistItem, PropertyInfo)
  - SetPropertyValue (Integer, interfaceName, attributeName)
  - SetPropertyValue (Integer, PropertyInfo)

- Could potentially raise an exception, so be ready to catch.

Generic Property Access – Set

- For unitted attributes, you have to use UOMMgr to parse the value input by user using ParseUnit and get the value in DB Units. To be able to use UOMMgr, you need to have the UnitType, which is available in the PropertyInfo.

- To get information about a Property, use **GetPropertyValue** on BusinessObject supplying the Interface and attribute names to get a **PropertyValue** object, which internally contains the **PropertyInfo** object.

- The **PropertyInfo** object contains all the information about the property. (we discussed that in our Metadata Manager session)

  Name DisplayName CodelistInfo PropertyType UOMType and more

- If **UOMType** is non zero, it means it is a unitted attribute. A value keyed in by user for a unitted attribute must be parsed and converted to "internal" units [standard units, called Database Units (abbreviated DBU)] for persistence in the database.

Generic Property Access – PropertyInfo

- We use **ParseUnit** method on **UOMMgr** to interpret user's keyed in value for an attribute.
- **ParseUnit** takes **UOMType** (available on PropertyInfo) and keyed in value to return the DBU value.
  - If value is specified without units, then, as a general guideline you have to interpret the input value as in active Default units. ParseUnit ( ) takes care of this as it is aware of the active settings for each unit type.
  - exceptions to this "interpreting" guideline could be for cases which your command decides to interpret such attribute's value in a specific unit irrespective of the active unit for its type.
  - For example, in your command, you may want to interpret user's Insulation Thickness keyin as "inches" irrespective of active distance units.
- Beware of **Complex** properties (like position on a ControlPoint). Such properties cannot be Get/Set using GetPropertyValue and SetPropertyValue methods. This is not a problem because such Complex properties are made up of normal properties on the same object and are exposed, so you can access those properties.

Generic Property Access – Set …

- The property you are trying to set may not stay on, in some occasions like below
  - If BusinessObject's logic calculates and sets it and doesn't allow user to set that property.
  - It is a Hierarchical Codelisted attribute, and the parent Codelist attribute has been modified. Triggering an internal setting of the Child Codelist attribute.
  - It is a Hierarchical Codelisted attribute, and the child Codelist attribute has been modified, this auto-setting the parent Codelist attribute value.
  - In some situations, the compute may invoke a solver which has busines logic to calculate and set this property, eg. on an Object which has a NameRule assigned, setting Name using generic access doesn't stay as the NameRule semantic triggers on compute and reassigns a name by rule. (It is for this purpose there is a SetUserDefinedName method.)

GetAllProperties

- This method returns a ReadOnlyCollection of PropertyValue objects.
- It is better to avoid calling this method as it may have a performance hit of unnecessarily querying and preparing a list of all properties of the object.

S3D .net API

LAB

-

Generic Property Access Command

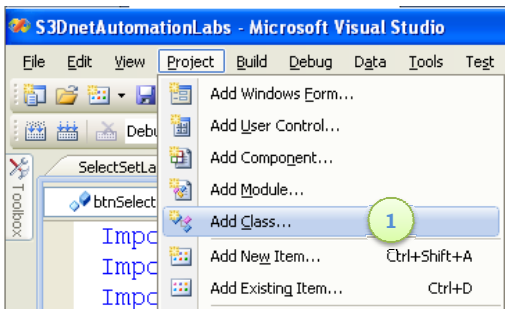
LAB - Overview

- In this LAB we will learn
  - Writing a Custom command in VB.net which displays a form enabling the user to see and change the Maximum Operating Temperature and Pressure of a Pipe run.
    - Create a Modal Command
    - Get the Select Set and examine its content
    - Get an Object from SelectSet
    - Use generic methods to get and set any property of a business object.
    - Use .SupportsInterface method on BusinessObject to determine if the object is a PipeRun
    - Use UOM service to
      - format an attribute displayed to user.
      - Parse an attribute value keyed in by user.
    - Use Transaction Manager to commit a transaction.

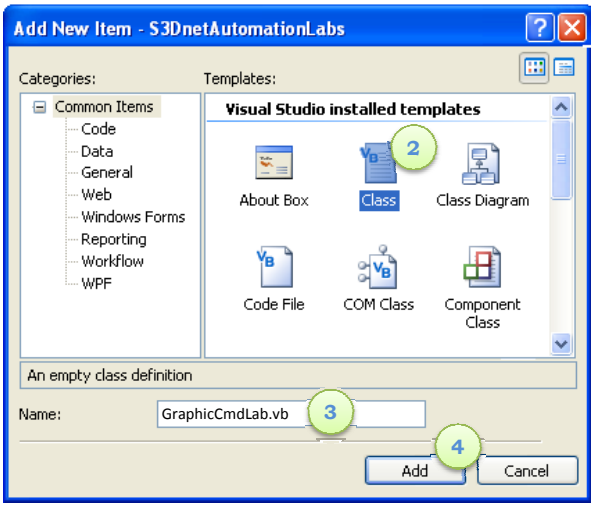
Create a new Command Class

1. Inside the Visual Studio Project, Click Project > Add Class… menu item.
2. Select **Class** from the Templates.
3. Specify the Name of the class file as desired.

   We choose "TempAndPressureMod.vb"



4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.

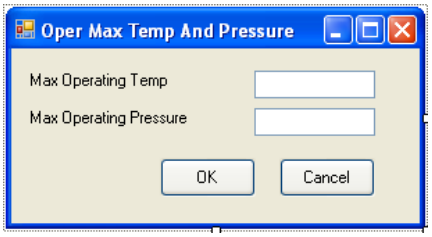


Create a new Form for Command

Our command needs a form as we need to do some UI interaction.

Create and add a form to the project.

1. Inside the Visual Studio Project, Click Project > Add Windows Form … menu item.
2. Select **Windows Form** from the Templates.
3. Specify the Name of the Form file as "frmTempAndPressureMod.vb".
4. Set the form Text property to "Oper Max Temp and Pressure" – this is what will show as the form title.
4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.



5. Add 2 labels, 2 Text Boxes, and 2 buttons as shown. We have named the Text Boxes **'txtTemp'** and **'txtPressure'**, and the buttons **'btnOK'** and **'btnCancel'**.

Implementation

Our command will Inherit from BaseModalCommand and has a form.

In OnStart( ) we first obtain a reference to the Select Set and examine its contents.

If exactly one Pipe run is selected we show the form Modally (using **ShowDialog()**).

If not, we'll display a message to the user prompting to select a single Pipe Run before invoking this command.

```
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle

Public Class TempAndPressureMod
    Inherits BaseModalCommand

    Private m_frmCmdForm As frmTempAndPressureMod

    Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)

        'get select set and make sure we have exactly one pipe run
        Dim oSelSet As SelectSet = ClientServiceProvider.SelectSet
        If oSelSet.SelectedObjects.Count <> 1 Then
            MsgBox("You need to select a single Pipe Run before running the command.")
        Else
            If (oSelSet.SelectedObjects.Item(0).SupportsInterface("IJRtePipeRun")) Then
                m_frmCmdForm = New frmTempAndPressureMod
                m_frmCmdForm.ShowDialog()
            Else
                MsgBox("You need to select a single Pipe Run before running the command.")
            End If
        End If

    End Sub

End Class
```


Implementation of Form logic

Declare a private variable to hold a reference to a UOMManager since we'll be working with unitted values.

Also we need a reference to the selected Pipe Run.

Note that we are referring to the PipeRun as BusinessObject since we'll be using generic methods to access properties.

We don't need to access any specific properties made available on PipeRun class. So accessing a PipeRun as a BusinessObject is more than enough for us.

```
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle
Imports Ingr.SP3D.Common.Middle.Services

Public Class frmTempAndPressureMod

    Private m_oUOM As UOMManager
    Private m_oPipeRun As BusinessObject
```

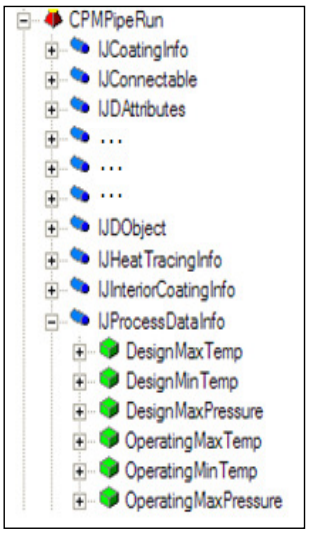
Implementation of Form logic …

In the form Load() event we initialize the references. The UOMManager from MidleServiceProvider and the PipeRun from the SelectSet.

We Get, Format and Display the OperatingMaxTemp & OperatingMaxPressure values.

Note that properties are defined by Interface & Attribute names. This information is available in the Metadata.

```
Private Sub frmTempAndPressureMod_Load(ByVal sender As System.Object, ByVal e As System.Eve
    Dim oTempVal As PropertyValueDouble, oPressureVal As PropertyValueDouble

    'get UOM
    m_oUOM = MiddleServiceProvider.UOMMgr

    'get select set
    Dim oSelSet As SelectSet = ClientServiceProvider.SelectSet

    'get the selected Pipe Run
    m_oPipeRun = oSelSet.SelectedObjects.Item(0)

    'get the Pipe Run's temp and pressure
    oTempVal = m_oPipeRun.GetPropertyValue("IJProcessDataInfo", "OperatingMaxTemp")
    oPressureVal = m_oPipeRun.GetPropertyValue("IJProcessDataInfo", "OperatingMaxPressure")

    'format and display. Dont format if value is not set yet.
    If (Not oTempVal.PropValue Is Nothing) Then _
        txtTemp.Text = m_oUOM.FormatUnit(UnitType.Temperature, oTempVal.PropValue)
    If (Not oPressureVal.PropValue Is Nothing) Then _
        txtPressure.Text = m_oUOM.FormatUnit(UnitType.ForcePerArea, oPressureVal.PropValue)
End Sub
```



Implementation of Form logic …

Double-click on the OK button in the form designer to add a button click handler. Inside we'll parse the user input to get the new values, we'll set those new values to the Pipe Run, and then we'll commit our changes

```
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    Dim dTemp As Double
    Dim dPressure As Double

    'parse the user entered values
    Try
        dTemp = m_oUOM.ParseUnit(UnitType.Temperature, txtTemp.Text)
    Catch
        MsgBox("Please enter valid temperature")
        Exit Sub
    End Try

    Try
        dPressure = m_oUOM.ParseUnit(UnitType.ForcePerArea, txtPressure.Text)
    Catch
        MsgBox("Please enter valid pressure")
        Exit Sub
    End Try

    'set new values
    m_oPipeRun.SetPropertyValue(dTemp, "IJProcessDataInfo", "OperatingMaxTemp")
    m_oPipeRun.SetPropertyValue(dPressure, "IJProcessDataInfo", "OperatingMaxPressure")

    'commit
    ClientServiceProvider.TransactionMgr.Commit("Modify Temperature and Pressure Command")
    Me.Close()

End Sub
```

Implementation of Form logic …

Similarly, we'll add a handler for the Cancel button. There we will simply abort our transaction and close the form.

```
Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnCancel
    'abort and close
    ClientServiceProvider.TransactionMgr.Abort()
    Me.Close()

End Sub
```

When the Form Closes, the Form.ShowDialog method call in the OnStart() implementation returns and then the execution continues in OnStart(). When OnStart all logic is completed, it returns back control to the system (Command Manager) and it examines the Modal property, and finding it as Modal=True, it calls stop command and stops the command eventually.

Our command implementation is complete now, and you can Run/Debug the command.

S3D .net API

Generic Relationship Access

-

Traverse Relationships

Generic Relationship Access – Traverse Relationships

BusinessObject, the base class of all the SmartPlant objects, provides ***Generic Relationship Access*** mechanism to access any related Object to.

To access related objects, all you need to know is the **RelationshipName** and the **RoleName** collection you are interested to access.

Given Name of Relationship one can get **RelationshipInfo** from **MetadataManager**. This is just the relationship pattern definition. Once that is coupled with a Business Object Instance, then you can access that Business object's related objects for that relationship pattern.

The returned Role Collection objects need not be in the same connection as the source object (eg PartOcc (Model) → Part definition (Catalog)).

To know about an object's possible relationship patterns, one could use **MetadataBrowser** and look up under each interface which the class implements to find the relationship pattern name. Inside 3D, one could also use **Repository Browser** (**Ctrl+Shift+R)** after selecting object(s) and browse the existing relationships it shows.

End Users cannot create relationship instances. They can only ***traverse*** existing relationship instances created by the system.

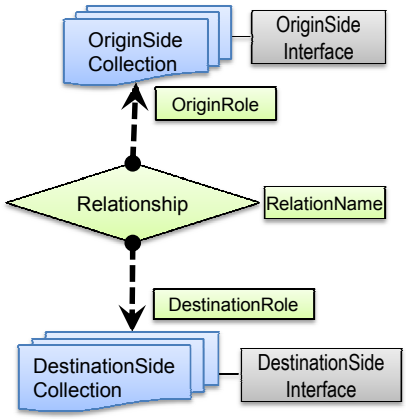
For people who know SP3D COM object model, this is similar to using DRelationHelper, DCollectionHelper, DRelationshipHelper, IJAssocRelation and other interfaces/classes.

Generic Relationship access

BusinessObject.**GetRelationship** (relationName, roleName)
BusinessObject.**Relationships** ( )

RelationCollection
- **Links** as ReadOnlyCollection(Of Link)
- **RelationshipInfo** as RelationshipInfo
- **Side** → Origin or Destination
- **Source** as BusinessObject
- **TargetObjects** as ReadOnlyCollection(Of BusinessObject)

Link
- **Name** as String
- **Target** as Business Object



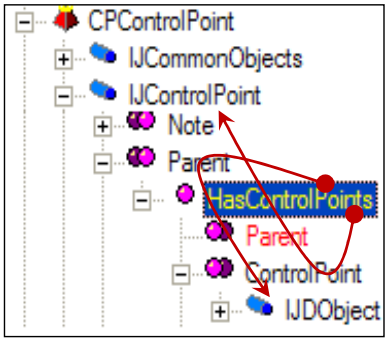
Note: MetadataBrowser shows roles flipped "Pointing to" interpretation

Interpretation: From *IJControlPoint Interface*, traversing *HasControlPoints relation* to get objects on *Parent* role gets you *IJDObject*, the Parent of the ControlPoint.

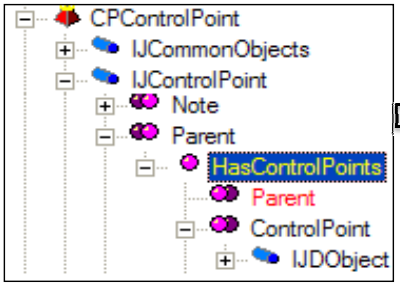
i.e. Interface → Role gives you Collection of other side objects

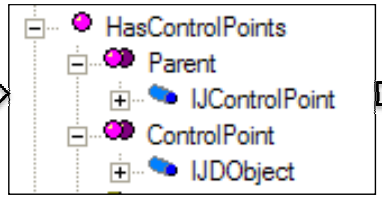
IJControlPoint → "Parent" ==> Collection of IJDObject type objects

TIP - Using the GetRelationship (relationName, roleName) method, you have to take a U-Turn, as shown in picture.

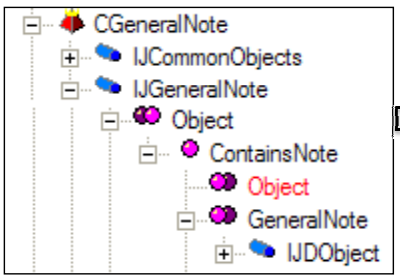


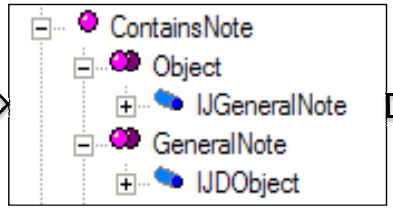
Examples of some Common Relationships



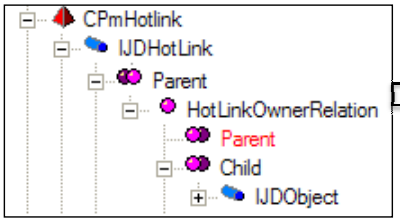


Object has ControlPoints
Parent Role
→ the Object
ControlPoint Role
→ the Control Point





Object Contains Note
Object Role
→ the Object
GeneralNote Role
→ the Note



HotLink to OwnerRelation
Parent Role
→ the Object
Child Role
→ the Hyperlink

Generic Relation Access – Get Related Objects

Three ways to access related objects of a BusinessObject.

- BusinessObject.GetRelationship (relationName, roleName) as RelationCollection
  - This is the most useful case. Just pass the **relationName** and **roleName** and this method returns you the RelationCollection.
- BusinessObject.GetRelationship (relationshipInfo, roleName) as RelationCollection
  - Use this when you already have a **RelationshipInfo** object. Method returns RelationCollection
- BusinessObject.Relationships ( ) as ReadonlyCollection (Of RelationCollection)
  - Use this only when you want to access all relationship patterns and get all related objects. This Property returns Collection of RelationCollections.
- Once you have RelationCollection, just use the **TargetObjects** property on it to access the objects. If the related objects are related through named relationships, you can use the **Links** property, to get those.

S3D .net API

LAB

–

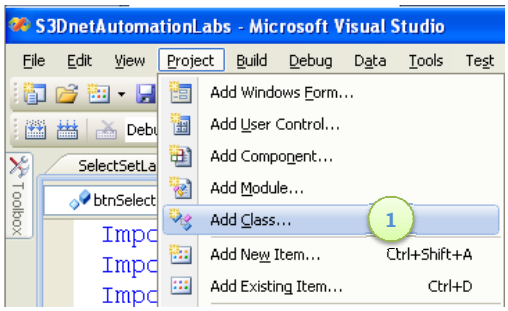
Generic Relationship Access

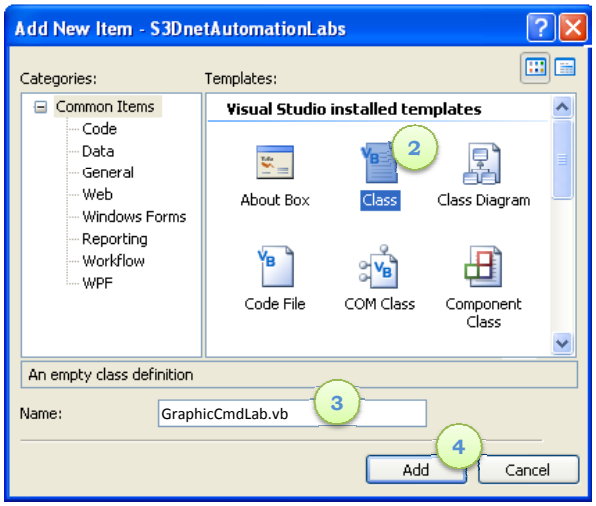
LAB - Overview

- In this LAB we will learn
  - Writing a Custom command in VB.net which will traverse relationship to access a collection of related objects of a given object.
  - We will deal with the following topics in the LAB.
    - Access SelectSet Service
    - Use EnableUISettings property to indicate that we need NonEmptySelectSet for the command to run.
    - Examine if an object is of a given type.
    - Use MetadataBrowser to understand Relationships, Relationship Information – Roles, Cardinality, Ends, Interfaces participating in relationships.
    - Access a BusinessObject's Name using BusinessObject.ToString( ) method.
    - Generic Relationships Access – Get Related objects of a Given Object.

Create a new Command Class

1. Inside the Visual Studio Project, Click Project > Add Class… menu item.
2. Select **Class** from the Templates.
3. Specify the Name of the class file as “RelationshipsLab.vb”
4. Click Add. A new file gets created and added to our project. It contains class implementation which we will finalize.





Implementation

Our RelationshipsLab command will Inherit from BaseModalCommand.

```
Imports System.Collections.ObjectModel
Imports System.Text
Imports Ingr.SP3D.Common
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Middle

Imports Ingr.SP3D.Systems.Middle
Imports Ingr.SP3D.Route.Middle

Public Class RelationshipsLab

    Inherits BaseModalCommand
```

We need to specify in EnableUIFlags() that our command needs a non empty Select Set

```
Public Overrides ReadOnly Property EnableUIFlags() As Integer
    Get
        Return EnableUIFlagSettings.NonEmptySelectSet
    End Get
End Property
```

Implementation …

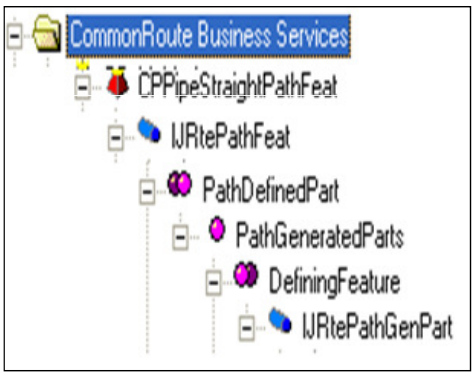
The only other method we need to override is **OnStart( ) –** in here, first we need to get SelectSet service and make sure we have only a single ***RouteFeature*** object selected

```
oSelectSet = Client.Services.ClientServiceProvider.SelectSet
If (oSelectSet.SelectedObjects.Count <> 1) Then GoTo ExitWithMessage
If Not (TypeOf oSelectSet.SelectedObjects.Item(0) Is RouteFeature) Then GoTo ExitWithMessage
```

We then need to get the reference to that Feature and access its related Route Parts collection.

We use ***RouteFeature*** type for the ***oFeature*** variable. We could also use BusinessObject as it is the base class of RouteFeature class.

We'll use the Metadata browser to get Relationship details (relationship & role names) which we will need to call GetRelationship( ) method.




```
oFeature = oSelectSet.SelectedObjects.Item(0)
oRelationColl = oFeature.GetRelationship("PathGeneratedParts", "PathDefinedPart")
```


Implementation …

We will the clear the Select Set, loop through the collection of related parts and add each part to the Select Set and each part name to a string.

```
' Clear the SelectSet, and then add the Parts found to Select Set.
oSelectSet.SelectedObjects.Clear()

Dim i As Long, oBO As BusinessObject
For i = 0 To oRelationColl.TargetObjects.Count - 1
    oSelectSet.SelectedObjects.Add(oRelationColl.TargetObjects.Item(i))
    oBO = oRelationColl.TargetObjects.Item(i)
    strMessage.Append(oBO.ToString & vbCrLf)
Next i

MsgBox(strMessage.ToString(), , "Parts on Selected Feature")
```

The final OnStart() code is:

Implementation …

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)
    Dim oSelectSet As SelectSet = ClientServiceProvider.SelectSet

    If (oSelectSet.SelectedObjects.Count <> 1) Then GoTo ExitWithMessage
    If Not (TypeOf oSelectSet.SelectedObjects.Item(0) Is RouteFeature) Then GoTo ExitWithMessage

    Dim oFeature As RouteFeature, oRelationColl As RelationCollection
    Dim strMessage As New StringBuilder()

    'object in SelectSet is RouteFeature, it participates in a relation with parts
    oFeature = oSelectSet.SelectedObjects.Item(0)
    oRelationColl = oFeature.GetRelationship("PathGeneratedParts", "PathDefinedPart")
    strMessage.Append(oRelationColl.TargetObjects.Count & " Parts related to selected Feature: " & vbCrLf)

    ' Clear the SelectSet, and then add the Parts found to Select Set.
    oSelectSet.SelectedObjects.Clear()

    Dim i As Long, oBO As BusinessObject
    For i = 0 To oRelationColl.TargetObjects.Count - 1
        oSelectSet.SelectedObjects.Add(oRelationColl.TargetObjects.Item(i))
        oBO = oRelationColl.TargetObjects.Item(i)
        strMessage.Append(oBO.ToString & vbCrLf)
    Next i
    MsgBox(strMessage.ToString(), , "Parts on Selected Feature")

    Exit Sub
ExitWithMessage:
    MsgBox("Select ONE Route Feature and run this command")

End Sub
```


Implementation …

- Our command implementation is complete now, and you can Run/Debug the command.

- We will also try some other relationships & roles.
- Look in Metadata Browser and try more relationships and roles and see what you can find.

S3D .net API

Step Commands

Step Commands - Overview

- Step Commands are a little advanced over Graphic Commands. They provide below capability
  - Organize a Command into Steps. Can specify Number of Steps.
  - Element Selection capability within the command in each step.
  - Each Step has various settings one can specify, eg this step's Prompt, ElementFilter criteria, Max Number of Elements, Objects Excluded from Locate, SmartSketch Settings, Locate Behavior, LocateHiliter (Color/Weight), HiliteLocated / Selected (whether Located / Selected Objects are Hilited).
  - Command inputs collected within a Step (SelectedBusinessObjects, Positions of Selected Objects) are organized within step.
  - Command maintains Current Step. Can be get/set.
  - React to Step Change events / SmartSketch events.
  - Some Command global behavior control and data (inputs collected)
    - Cumulative selection (All objects selected in all steps) is hilited or Not,
    - GloballyExcludeFromLocate (ObjectsList)
    - StopCommandOnRightClick – whether or not system stops command on right click.
    - Enabled or Not at a given time(get events or not)

Step Commands - Overview

- You can override Graphic View Manager Events, Keyboard events.
- **Step's StepFilter** can have criteria added eg.

  Step(0).**StepFilter.AddInterface** ("IJEquipment")

  Step(0).**StepFilter.AddLogicalOperator** (OR)

  Step(0).**StepFilter.AddInterface** ("IJShape")
- **StepFilter** → like a mini filter based on Interface/Class/BOC names and AND/OR/NOT operators. Can also setup to call a User Defined Function.
- Set it up using methods as above (Add**Interface** / **Class** / **BOC** / **LogicalOperator** / **Function**) and it only lets you select those in that step.
- Position calculation has SmartSketch support. Based on SS3D settings active in that step, position generated is solved to that constraints.
- Rest everything (including RibbonBar handling) is same as GraphicCommand.

Step Filter

- Step Filter can also be used outside of step commands, just for filtering objects using the Step Filter approach.
- Just new one, setup the Filter criteria and call the methods to Filter.
- **Apply (*BOCollection*)** → Filters "in-place" the objects in BOCollection
- ***CheckObject(BusinessObject*)** → returns whether/not the object satisfies the StepFilter definition.
- ***GetNumberOfObjectsThatPass(BOCollection*)** → returns number of objects passing the StepFilter Definition.
- **Clear()** → Clears the criteria set so far, useful if you want to set new criteria.

Using UserFunction in Step Filter Definition

- Using the **AddFunction** method, you can specify an object which is *instance* of a class implementing **IStepFilterFunction** which can filter elements using custom code (UserFunction).
- **IStepFilterFunction** interface has ***Evaluate*** (BusinessObject) Function [boolean] and ***Parameter*** property [object].
  - Parameter is useful to parameterize the UserFunction evaluation logic.
- Use the *StepFilter.AddFunction* passing the *instance*. If the *Evaluate* function uses the *Parameter*, then logic invoking StepFilter must set it on the *instance* before the *Evaluate* gets called. (i.e. StepFilter is evaluated)
- The system calls your *Evaluate* method passing the BusinessObject being evaluated by StepFilter in action. Your logic in *Evaluate* needs to analyze the **BusinessObject** and return true/false as appropriate.

Example – Using UserFunction in Step Filter Definition

- Lets say we have a command which has two steps,
- 1st step looks for PipeRun, and
- 2nd Step looks for PipeRuns with same NPD as the first one.

```
'Step ONE selects a PipeRun
Step(0).StepFilter.AddInterface ("IJRtePipeRun")
...
'Step 2 Selects PipeRuns with same NPD as the one selected by Step1
Step(1).StepFilter.AddInterface ("IJRtePipeRun")
Step(1).StepFilter.AddLogicalOperator (AND)
m_oFilterRunsByNPDFxn = New FilterForRunsByNPD
Step(1).StepFilter.AddFunction (m_oFilterRunsByNPDFxn)
...
' Init the parameter to the NPD of the PipeRun in Step1
m_oFilterRunsByNPDFxn.Parameter = oPipeRun1.NPD
```

```
Class FilterForRunsByNPD
  Implements IStepFilterFunction

  Private m_oParameter as Object
  Function Evaluate (oBO  as BusinessObject ) as Boolean
    if Not (Typeof oBO is PipeRun) then Return False
    Dim oRun as PipeRun = oBO
    ' Assumes the StepFilter using this has set the NPD as Parameter.
    Dim oNPD as NominalDiameter = Parameter
    Return (oRun.NPD.Size = oNPD.Size And oRun.NPD.Units = oNPD.Units)
  End Function

  Property Get Parameter () as Object
    return m_oParameter
  End Property
  Property Set Parameter (oParam as Object)
     m_oParameter = oParam
  End Property
End Class
```

Step Events

- Command can handle SelectionChange events within a Step – i.e. as and when elements are selected/deselected in a step.
- Command is can handle Step change events (***OnStep*** - ***Begin***, ***End)*** and Step Selection Change events ***(OnStep - SelectionChanged, PostSelectionChanged***) to drive the functionality of the command – move between steps and/or process element selection as it happens.
- Call UpdateStep() to reprocess the data of the step (selected Objects etc).

SmartSketch Usage in Step

- A Step's LocateBehaviour property can be set to one of these options – **Boreline / QuickPick / Fence / SmartSketch3D.**
- When Step's LocateBehaviour is set to SmartSketch3D
  - Each **Step** can setup its own SSK Settings.
  - For example, a Step may be specifically looking for "Point ON Line" or "Nozzle/Port Point" and nothing else. It could therefore turn these options alone to "ON" and rest everything to false. When active in such step, SSK would only solve for those constraints.

```
Steps(1).SmartSketchOptions.PointOnCurve = True
Steps(1).SmartSketchOptions.PointOnPort = True
```

  - Command's logic can handle the events raised by SSK (***OnObjectsLocated, OnSmartSketchPositionChange, OnSmartSketchConstraintConflict***).

S3D .net API

LAB

–

Step Commands

LAB - Overview

- In this LAB we will learn
  - Writing a Step Command in VB.net with 2 steps.
    - 1st step allows the user to select one or more equipments
    - Second step will place these equipments one by one at each left mouse click
    - In 2nd step, the user using the Right/Left Arrow keys to iterate through the step 1 selection can change the equipment currently being repositioned.

Implementation

Create a new class SmartCmdLab which inherits from **BaseStepCommand**.

Add a private member to keep track of the object currently being repositioned in the 2nd step.

```
Imports Ingr.SP3D.Common.Client
Imports Ingr.SP3D.Common.Client.Services
Imports Ingr.SP3D.Common.Middle
Imports Ingr.SP3D.Equipment.Middle
Imports System.Collections.ObjectModel

Public Class SmartCmdLab
    Inherits BaseStepCommand

    Private m_activeItemIdx As Integer
```

EnableUIFlags() – Command needs Active View

```
Public Overrides ReadOnly Property EnableUIFlags()
    Get
        Return EnableUIFlagSettings.ActiveView
    End Get
End Property
```

OnStop() : Call Abort() → do not leave any uncommitted changes for next command.

```
Public Overrides Sub OnStop()
    ClientServiceProvider.TransactionMgr.Abort()
End Sub
```

Note, the System automatically deals with the Command's .Enabled Property. You don't have to set it explicitly in your OnSuspend/OnResume implementation.

You only need to unsubscribe/subscribe to any events you want.

Next, OnStart()
- Set number of steps
- Set properties for each step
- Enable and set active step

Implementation

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)

  StepCount = 2

  'used only for steps where UseStepHiliterForSelection = False
  SelectHiliterColor = ColorConstants.RGBBlue
  SelectHiliterWeight = 2

  '1st step
  Steps(0).Prompt = "Select Equipment(s) with Fence. Press UpArrow Key when ready to specify positions."
  Steps(0).LocateBehavior = StepDefinition.LocateBehaviors.Fence
  Steps(0).StepFilter = New StepFilter ' Required in v2009.1
  Steps(0).StepFilter.AddInterface("IJSmartEquipment")
  Steps(0).UseStepHiliterForSelection = True ' We use Step Hiliter
  Steps(0).HiliteSelected = True
  Steps(0).SelectHiliterColor = ColorConstants.RGBWhite
  Steps(0).SelectHiliterWeight = 2

  '2nd step
  Steps(1).Prompt = "Specify Eqp Position; Left/Right Arrow = Prev/Next Eqp"
  Steps(1).LocateBehavior = StepDefinition.LocateBehaviors.SmartSketch3D
  Steps(1).UseStepHiliterForSelection = False
  Steps(1).SmartSketchOptions.EvaluateCenterpointOn = True
  Steps(1).SmartSketchOptions.CenterPoint = True
  Steps(1).SmartSketchOptions.ReferenceAxisAlignment = True
  Steps(1).SmartSketchOptions.Intersection = True
  Steps(1).SmartSketchOptions.MidPoint = True
  Steps(1).SmartSketchOptions.PointOnCurve = True
  Steps(1).SmartSketchOptions.PointCreator = Nothing

  ClientServiceProvider.TransactionMgr.Abort()
  CurrentStepIndex = 0 'set active step to 0
  Enabled = True
```

Note: In v2009.1, some of these options have been Obsoleted as eqvt. options with better descriptive names are provided. Follow the Obsolete text displayed in the compiler warnings to know the eqvt new option. Old options will work until removed permanently (usually in later version).

For example .EvaluateCenterPointOn = True is now replaced with .CenterPoint = True

Implementation

OnMouseMove() – if 2nd step, reposition currently modified Eqp, Compute to see the changes

```
Protected Overrides Sub OnMouseMove(ByVal view As Ingr.SP3D.Common.Client.Services.GraphicView, By
  Try
    If CurrentStepIndex = 1 Then

      If (m_activeItemIdx > 0 And m_activeItemIdx <= Steps(0).SelectedBusinessObjects.Count) Then
        Dim oEqp As Equipment = Steps(0).SelectedBusinessObjects(m_activeItemIdx - 1)
        oEqp.Origin = position

        ClientServiceProvider.TransactionMgr.Compute() ' Shows Dynamic Position
      End If
    End If

  Catch ex As Exception
    MsgBox("" & ex.Message, , "Error in MouseMove")
  End Try

End Sub
```

OnMouseDown() – if 2nd step, and Left Click, advance to next Eqp

```
Protected Overrides Sub OnMouseDown(ByVal view As Ingr.SP3D.Comm

  If e.Button = MouseButtons.Left And CurrentStepIndex = 1 Then
    PreviousOrNext(1) ' Set active item = next
  End If
End Sub
```

RightClick to stop command – taken care by Default in Step Commands.

Implementation

OnKeyDown()
- Left/Right Arrow – change the currently modified equipment
- Up Arrow – change active command step from 1st to 2nd step

```
Protected Overrides Sub OnKeyDown(ByVal e As Ingr.SP3D.Common.Client.BaseGraphicCommand.K

  If e.KeyValue = System.Windows.Forms.Keys.Left Then 'Goto Previous Equipment
    PreviousOrNext(-1) ' Set active item = previous

  ElseIf e.KeyValue = System.Windows.Forms.Keys.Right Then ' Goto Next Equipment
    PreviousOrNext(1) ' Set active item = next index

  ElseIf e.KeyValue = System.Windows.Forms.Keys.Up Then 'Key For setting "Position" Step
    Dim nSelected = Steps(0).SelectedBusinessObjects.Count
    If (CurrentStepIndex = 0) And Not (nSelected = 0) Then
      CurrentStepIndex = 1 ' goto "Select Position" step
      m_activeItemIdx = 1 ' reset to 1
      PreviousOrNext(0) ' Set active item = active index
    End If

  End If

End Sub
```

Implementation

Helper Routines:

- OnFinish() – commits and exits

- PreviousOrNext – advances to the next/previous equipment in the 1st step selection.

```
Private Sub OnFinish()
    ClientServiceProvider.TransactionMgr.Commit("Step Command Commit")
    StopCommand()
End Sub
Private Sub PreviousOrNext(ByVal delta As Integer)

    m_activeItemIdx = m_activeItemIdx + delta
    With Steps(0).SelectedBusinessObjects
        ' allow 'parking at '0' and 'n+1' so user can
        ' press left/right arrow keys to go back/forward.
        If (m_activeItemIdx < 0) Then
            m_activeItemIdx = 0
        ElseIf (m_activeItemIdx > .Count + 1) Then
            m_activeItemIdx = .Count + 1
        End If
    End With

End Sub
```

Our command implementation is complete now, and you can Run/Debug the command.

Improvements

We will do some improvements to the command.

1. Show a Line indicating the Move, and also direction if along active XYZ axes.
2. Commit Each Eqp position change individually (on MouseDown alone).
3. Achieve ONE undo of the entire operation of moving all equipment.

```
Public Class SmartCmdLab
  Inherits BaseStepCommand

  Private m_activeItemIdx As Integer ' Which Eqp being moved
  Private m_oMoveIndicator As Line3d ' Temporary line to indicate move.
  Private m_oMoveHiliter As GraphicViewHiliter  ' To Hilite Eqp being moved and Direction
  Private m_sCommitString As String  ' To get single UNDO for multi commits.
```

We add some class variables to hold the MoveIndicator Line, Hiliter to show the Move Indicator line, and commit string to use.

```
  ' Create & Setup the Move Hiliter
  m_oMoveHiliter = New GraphicViewHiliter
  m_oMoveHiliter.Color = ColorConstants.RGBYellow
  m_oMoveHiliter.Weight = 3
  m_oMoveHiliter.LinePattern = HiliterBase.HiliterLinePattern.Dotted

  m_sCommitString = "Reposition Multiple Equipment"

End Sub
```

*In the End Of **OnStart**, we Create and Configure our Move Hiliter*

Improvements

We modify PreviousOrNext( ) routine which shifts from one item to another – changes are to –
***1)** set the AssistingPosition, **2)** create Move Indicator line if not done yet, **3)** set its Start point.*

```
Private Sub PreviousOrNext(ByVal delta As Integer)

  m_activeItemIdx = m_activeItemIdx + delta
  ' allow 'parking at '0' and 'n+1' so user can press left/right arrow keys to go back/forward.
  If (m_activeItemIdx < 0) Then
    m_activeItemIdx = 0
  ElseIf (m_activeItemIdx > Steps(0).SelectedBusinessObjects.Count + 1) Then
    m_activeItemIdx = Steps(0).SelectedBusinessObjects.Count + 1
  End If

  If (m_activeItemIdx > 0 And m_activeItemIdx <= Steps(0).SelectedBusinessObjects.Count) Then
    ' Get Eqp being moved
    Dim oEqp As Equipment = Steps(0).SelectedBusinessObjects(m_activeItemIdx - 1)

    ' Get its Origin
    Dim oPoint As New Position(oEqp.Origin)

    'Set it as StepCommand's SmartSketch's AssistingPosition
    SmartSketch.AssistingPosition = oPoint

    'Create Move indicator Line if not done already
    If m_oMoveIndicator Is Nothing Then m_oMoveIndicator = New Line3d(oPoint, oPoint)
    'Set Move Indicator's Start Point (End Point set in MouseMove)
    m_oMoveIndicator.StartPoint = oPoint

  End If
End Sub
```

We get current origin of Equipment being moved and set it as AssistingPosition for SmartSketch

We create the Move Indicator Line if not done yet, and set its Start Point to the Eqp's Origin.

Improvements

On MouseMove, after setting the Equipment's Origin and Computing, we set it as the end Point of the Move Indicator line – our move Indicator is finalized for this position now, and we add it to the move Hiliter

```
Protected Overrides Sub OnMouseMove(ByVal view As Ingr.SP3D.Common.Client.Services.GraphicView, By
  Try
    If CurrentStepIndex = 1 Then

      If (m_activeItemIdx > 0 And m_activeItemIdx <= Steps(0).SelectedBusinessObjects.Count) Then
        Dim oEqp As Equipment = Steps(0).SelectedBusinessObjects(m_activeItemIdx - 1)
        oEqp.Origin = position
        ClientServiceProvider.TransactionMgr.Compute() ' Shows Dynamic Position

        m_oMoveIndicator.EndPoint = position ' Set the MoveIndicator line's endpoint
        m_oMoveHiliter.HilitedObjects.Clear() ' Clear The Move Hiliter
        m_oMoveHiliter.HilitedObjects.Add(m_oMoveIndicator) ' Add Moveindicator to Hiliter
      End If
    End If

  Catch ex As Exception
    MsgBox("" & ex.Message, , "Error in MouseMove")
  End Try

End Sub
```

```
Protected Overrides Sub OnMouseDown(ByVal view As Ingr.SP3D.Common.Client.Services.Graph

  If e.Button = MouseButtons.Left And CurrentStepIndex = 1 Then
    ClientServiceProvider.TransactionMgr.Commit(m_sCommitString) ' Commit each Eqp Move
    m_sCommitString = "" ' Do subsequent commits with Blank string, gets single UNDO.
    PreviousOrNext(1) ' Set active item = next
  End If
End Sub
```

Improvements

On KeyDown, for Left/Right keys we abort the transaction to forget uncommitted changes (from MouseMove)

```
Protected Overrides Sub OnKeyDown(ByVal e As Ingr.SP3D.Common.Client.BaseGraphicCommand.K

  If e.KeyValue = System.Windows.Forms.Keys.Left Then 'Goto Previous Equipment
    ClientServiceProvider.TransactionMgr.Abort() ' forget uncommitted changes
    PreviousOrNext(-1) ' Set active item = previous

  ElseIf e.KeyValue = System.Windows.Forms.Keys.Right Then ' Goto Next Equipment
    ClientServiceProvider.TransactionMgr.Abort() ' forget uncommitted changes
    PreviousOrNext(1) ' Set active item = next index

  ElseIf e.KeyValue = System.Windows.Forms.Keys.Up Then 'Key For setting "Position" Step
    If (CurrentStepIndex = 0) And Not (Steps(0).SelectedBusinessObjects.Count = 0) Then
      CurrentStepIndex = 1 ' goto "Select Position" step
      m_activeItemIdx = 1 ' reset to 1
      PreviousOrNext(0) ' Set active item = active index
    End If

  End If
End Sub
```

Our command improvements are complete now, and you can Run/Debug the command.

S3D .net API

API Commonality, Discoverability, Documentation, Examples …

Common Features, Paradigms, Approaches followed in the API

Business Object Creation & Modification – Commonality

- Several Applications within S3D offer API (Constructors, Methods, Properties) to create / modify Business Objects types that Application handles.

  Common App → Filters, Notes, ControlPoint, WBSProject, WBSItem …

  Systems & Specs → System types (Area, Unit, Generic & Discipline Specific Systems)

  Equipment → Equipment, Equipment Component, Shapes, Nozzles

  Route → PipeRuns, Routing new runs, Inserting Components/Instruments.

  Structure → Placing new Linear members, Curved members, slabs etc.

  Hangers → Placing Hangers & Supports on Routed Parts, Add Support Parts etc.

  Grids → Creating Coordinate Systems, Grids – Grid & Elevation Planes, Gridlines.

  Space → Create Space Folders, Spaces, Zones.

  …

- The modification aspect in ***Application Specific API*** is in addition to the ***Generic Property Access*** mechanism which facilitates modifying properties of any Business Object in a *generic* manner, without *specifically* using the Application specific specialized BusinessObject types.

Business Object Creation & Modification – Commonality

- For example one can modify a Pipe Along Leg Feature's **ShortCode** property using
  1) *generic* API, by referring to it as a *generic* BusinessObject, *or*
  2) *specific* Application API, by referring to it as a (RoutePipe) Application specific PipeAlongLegFeature Object.

- Both the approaches do the same thing - set its ShortCode to "Gate Valve".

Assume SelectSet has ONE AlongLeg Pipe Feature object in it (say a Valve).

Generic Property Access API

```
Dim oBO As BusinessObject = oSelectSet.SelectedObjects(0)
oBO.SetPropertyValue ("IJRtePipePathFeat", "Type", "Gate Valve")
TransactionMgr.Commit ("Change ShortCode – Generic Property Access API")
```

Application (RoutePipe) Specific Property Modification API

```
Dim oPipeAlongLegFeature As PipeAlongLegFeature = oSelectSet.SelectedObjects(0)
oPipeAlongLegFeature.ShortCode = "Gate Valve"
TransactionMgr.Commit ("Change ShortCode – Using RoutePipe application API")
```


Business Object Creation Paradigm - Constructors

- For Business Object Creation in S3D .net API, the paradigm is **Constructors**
- Each Application's API provides
  - Constructors for all those objects end users are likely to create.e.g,
    - Route → PipeRun
    - Equipment → Equipment, Component, Shape, Nozzle creation.
  - For different variations of constructing business Objects, overloaded constructors are provided, which take different kinds of arguments to suit different usage scenarios.

```
Dim oEqp as Equipment
'This constructor uses Part Number as Parameter
oEqp = New Equipment ("HCPump05 8""x4""-E", oParentSys)
'This constructor uses Catalog Part as Parameter
oEqp = New Equipment (oCatalogEqpPart, oParentSys)
```

Business Object Creation Paradigm – Constructors …

Transient vs Persistent Object Construction.

- Certain Objects can be created either as persistent or temporary objects : for example Geometry objects like Lines, Spheres, Planes etc are created as
  - persistent objects, as created inside symbols, and
  - transient (temporary) objects, as created for graphic calculation/display hilite purposes but not for database persistence,

For such objects the API offers offer two variants of Constructors,

- One which takes SP3DConnection as a Parameter and
- Another without SP3DConnection as a Parameter.

Example, lets see **Circle3D** Constructors

Circle3D (Position1, Position2, Position3) 'Transient Circle by 3 points

Circle3D (SP3DConnection, Position1, Position2, Position3) ' Persistent Circle by 3 points

Circle3D (centerPosition, normalVector, dRadius) 'Transient Circle by center, Plane, Radius

Circle3D (SP3DConnection, centerPosition, normalVector, dRadius) 'Persistent Circle by center, Plane, Radius

S3D .net API Discoverability

- There are so many Constructors offered to make it simple for end user to use in different situations and inputs.
- How does end user find out all variations available, what parameters to pass …
- Even more … There are so many Classes, Constructors, Methods, Properties dealing with different Business Objects. How can end user deal with this?
- Yes – It is practically not possible to list all of them in a training document – difficult to search, organize, link-up, maintain etc.
- How to solve this ?
- Well, the answer lies in standard approach followed for **S3D .net API** development, certain tools / framework from Microsoft Visual Studio and ***.net*** Framework come to our rescue here.
- They make it a whole lot simpler, quicker, manageable, searchable and provide uptodate information.

S3D .net API Discoverability …

1) **Code Intellisense** → Use the power of the dot. On any object/class/interface just typing "." gives offers you Intellisense help. On classes/objects/interfaces it lists methods/properties, on methods it shows method parameters expected, for overloaded methods it indicates the number of overloads available and lets you choose one of it – and provides method syntax for each method overload.
2) **Help Yourself** ☺ → Use Help Documentation delivered with the API. Help is integrated into Visual Studio and all the S3D .net API Classes, Constructors, Methods, Properties, Interfaces etc usable by end users is documented, full examples as well as code snippets are provided.
3) **Use Object Browser** → Inside VB.net / C#, Object Browser can help browse through all of S3D .net API Classes, Constructors, Methods, Properties, Interfaces. This comes from their ***.net*** Metadata, and is the final truth. ☺. Even if something is not documented in Help, you will find it in here.
4) **High Level Overview** → Look at the S3D .net API Training Material, and the Programmers Guide.
5) **Look into Delivered Examples** → Look at $SP3DProgramming\<App>\SOM\Examples\ directory to find samples for each app.

S3D .net API

Application Functionality
–
Common Interfaces implemented by Several Discipline Objects

Application Functionality – Common Interfaces

Many Common methods/properties/info is available on BusinessObject, the baseclass of all S3D Business Objects.

Other Common interfaces which describe behavior implemented by different discipline objects are listed below.

INamedItem → NamedItem behavior
- Get Name
- SetUserDefinedName

INameRule → NameRule behavior
- Get/Set ActiveNameRule

ILocalCoordinateSystem → Origin/Orientation
- Origin
- Matrix
- XAxis, YAxis, Zaxis

ITransform → Transformable behaviour
- Transform (Matrix4x4)

IRange → Object's Range Information
- *Get* Range property

ISystem → System Parent Behavior
- Get SystemChildren
- AddSystemChild

ISystemChild → SystemChild
- Get/Set SystemParent

IAllowableSpecs → Allowed Specs mgmt
- Get/Set/Add/Remove/Replace/Reset Allowed Specs.

Application Functionality – Common Interfaces …

IWBSProject → WBS Project behavior
- AddToProject (WBSChild)
- RemoveFromProject (WBSChild)
- BelongsToProject (WBSItemChild) ??
- ProjectChildren ( ) → BOs in this WBSProject

IWBSParent → WBS Parent behavior
- *Get* WBSChildren
- AddWBSChild

IWBSChild→ WBS Child behavior
- *Get* WBSParent

IWBSItemChild→ WBS ItemChild behavior (exhibited by BOs which can be in a WBSItem)
- *Get* WBSItemParents

IWBSItem → WBSItem behavior
- AddWBSItemChild/Children
- RemoveWBSItemChild/Children
- *Get* WBSItemChildren
- ValidateChildren/Parents
- UpdateWBSItemAssignment

IPart → Part Behaviour
- PartClass
- PortDefinitions

IPort → Port behavior
- *Get* Connectable → The Part
- *Get* Connections → at this Port
- *Get* PortType

IConnection→ Connection behavior
- GetPorts
- FindPort

Application Functionality – Common Interfaces …

ICorrelatedObject → Corelated Object behavior
- Correlate(oBO, bUpdate, corelateStatus)
- UnCorrelate(oBO)
- Get Property/Topology Comparison
- IsValidDesignBasis
- MatchesDesignBasis
- SafeToDelete
- UpdateCorelationStatus

IAssembly → Assembly Parent behavior
- Get AssemblyChildren
- AddAssemblyChild

IAssemblyChild → Assembly Child behavior
- Get AssemblyParent

IConnectable → A 'Connectable' part behavior
- GetConnectedPorts
- GetConnectablePorts
- GetPorts

IDistributionConnection → Distribution Connection Information
- ConnectionParts
- Location

IDistributionPort → Distribution Port information (Fluid Flow related Ports)
- IsConnected
- Location
- Normal & Radial Vector properties

S3D .net API

Application Functionality

–

Core/CommonApp

Application Functionality – Core & CommonApp

Assembly : CommonMiddle.dll

Classes :

FilterFolder, Filter, CompoundFilter, SQLFilter, FilterParameter, Note, WBSProject, WBSItem, ControlPoint, CoordinateSystem, Point3D, Sketch3D, Sketch3DFeatures

GeometryClasses : Line, Circle, Ellipse, ComplexString, Sphere, Plane …

Filters

Provide programmatic Workspace Filter Functionality - Can create Transient (on the fly – use & throw) filters, or Persistent filters.

Can specify FilterDefinition → BOC, Hierarchy (System, Assembly, Analysis, Space, WBS, Permissiongroup, Volume_NamedSpace), Volume, Property, PropertyRange, Reference

Code Workflow : Create Filter, Set Filter Definition properties, Apply

Apply () 'returns objects satisfying this filter in active Connection.

Apply (SP3DConnection) 'returns objects satisfying this filter in given Connection.

FilterObject(oBO as BusinessObject) as Boolean '-> Check if Object passes

FilterObjects(oBOCollection as ReadOnlyCollection Of BusinessObject) as ReadOnlyCollection Of BusinessObject ' Returns Objects Passing through this Filter.

Compound Filters '→ Can combine two or more filters using AND, OR, NOT operators, and use Parenthesis as well.

SQL Filters → Can define by SQL query. Must be Persistent (in v2009/ 2009.1).

A small example of a Command using Filter

```
A command, demonstrating how to create a TransientFilter to get all
Pipe Straight Features under Selected System(s) which are longer than
1.0m and show their count, and Hilite them using a New Hiliter.
```

```
Public Overrides Sub OnStart(ByVal commandID As Integer, ByVal argument As Object)
  '  Create a transient Filter. For Persistent filter, use New Filter (ActiveModel)
  Dim oFilter As New Filter(), i As Long, oSelObjs As New Collection(Of BusinessObject)

  '  Prepare a Collection (Of BusinessObject) from SelectedObjects
  For i = 0 To ClientServiceProvider.SelectSet.SelectedObjects.Count - 1
    oSelObjs.Add(ClientServiceProvider.SelectSet.SelectedObjects.Item(i))
  Next
  ' This is how you can create a ReadOnlyCollection (Of X) from Collection (Of X)
  Dim oROBOColl = New ReadOnlyCollection(Of BusinessObject)(oSelObjs)

  ' Define Our Filter Definition Criteria
  '   uses System Hierarchy, specify selected systems, Include Nested Objects
  oFilter.Definition.AddHierarchy(HierarchyTypes.System, oROBOColl, True)
  '   uses ObjectType = PipeStraight.
  oFilter.Definition.AddObjectType("Piping\PipingFeatures\PipeStraight")
  '   Define a PropertyValue representing StraightPathFeat's Length = 1.0m
  Dim oPropValue As New PropertyValueDouble("IJRteStraightPathFeat", "Length", 1.0)
  '  uses Property .GE. oPropValue
  oFilter.Definition.AddWhereProperty(oPropValue, PropertyComparisonOperators.GE)
  Dim oFeats = oFilter.Apply() ' Apply to get objects satisfying Filter in DB

  ' Hilite them, demonstrates using a New Hiliter and Adding objects to Hiliter.
  Dim oHiliter As New Hiliter
  ' Set Hiliter properties, Color, Weight, etc as desired or accept default.
  oHiliter.HilitedObjects.Add(oFeats)
  MsgBox("Found " & oFeats.Count & " features")
End Sub
```

Note

Creation and Modification of Note on a BusinessObject

Constructors :

Note (BusinessObject) → Creates a note on Business Object.

Note (Port) → Creates a note on Port.

Note (ControlPoint) → Creates a note on ControlPoint.

Other Note Properties :

Note Properties (Purpose, Text etc) can be set using Generic Property Access.

ControlPoint

Creation and Modification of ControlPoint on a BusinessObject.

Behavior : Implements IPoint, ITransform, INamedItem

Constructors :

ControlPoint (BusinessObject, Position) → Creates ControlPoint on a BO at given location.

ControlPoint (BusinessObject, Position, Diameter) → Creates ControlPoint on a BO at given location with diameter.

ControlPoint (SP3DConnection, Position, Diameter) → Creates a standalone ControlPoint, in given connection.

ControlPoint (SP3DConnection, BusinessObject, Position, Diameter) → Creates Standalone ControlPoint, in given connection at given location with given diameter.

Properties :

Parent → the Owner Object if any of this ControlPoint.

Position → the Position of this ControlPoint.

Other Properties :

Other ControlPoint Properties (Type, SubType, Diameter) can be set using Generic Property Access.

WBSProject, WBSItem

Creation and Modification of WBSProject, WBSItem.

Behavior : Implement IWBSParent, IWBSChild, ICorrelatedObject, INamedItem

In addition,

WBSProject → Implements IWBSProject

WBSItem → Implements IWBSItem

Constructors :

WBSProject () → Creates a New WBS Project.

WBSItem (oWBSParent, exclusiveFlag, assignmentType) → Creates New WBS Item under given WBSParent with given details.

Properties / Methods :

See IWBSItem, IWBSParent, IWBSChild, IWBSProject for specific WBS related properties/methods.

Other Properties :

Other WBSProject/Item type Properties can be set using Generic Property Access.

S3D .net API

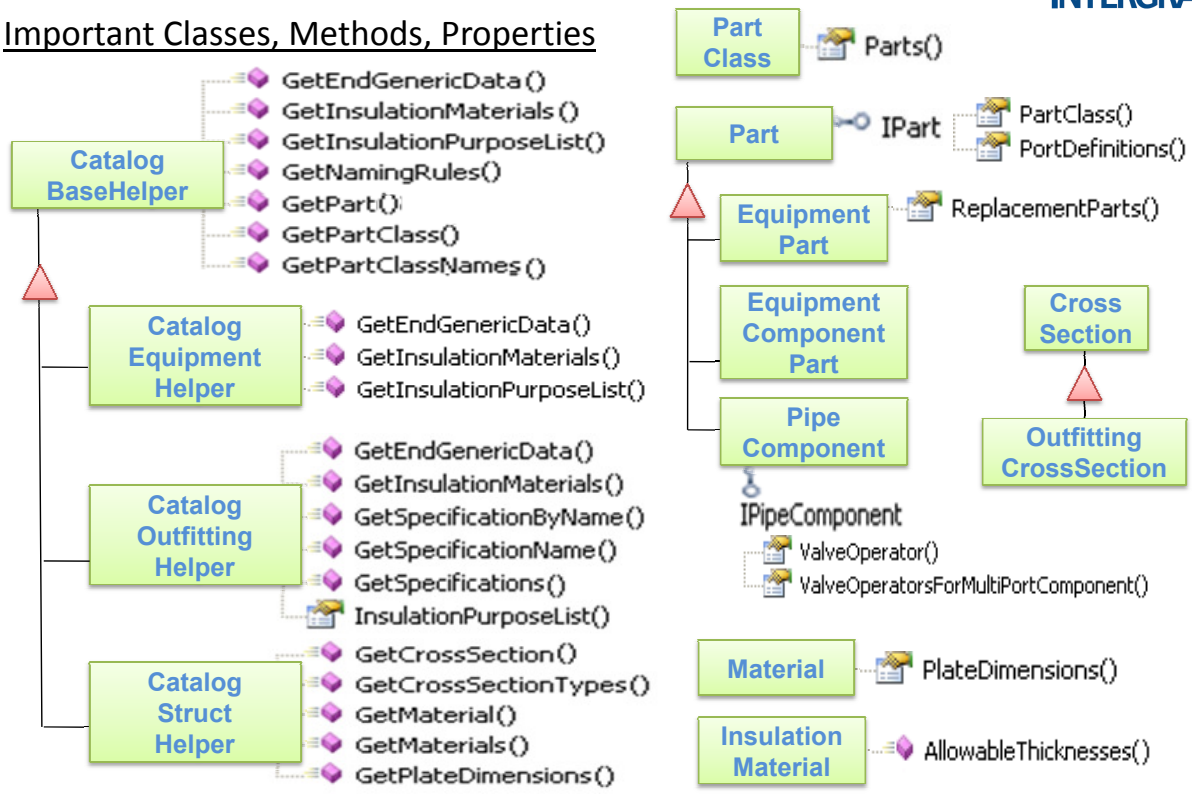
Application Functionality

–

Catalog

Application Functionality – Catalog

Important Classes, Methods, Properties



S3D .net API

Creating Standalone Applications

Standalone Apps

- S3D .net API allows one to write Standalone Apps, i.e. Automation outside of TaskHost executable.

- What kinds of Automation can you write in Standalone Application ?
  - Checking Automation – Check objects/data and take some action like generating a report file / output file.
  - Data Mining – Do some data processing / data mining on objects and related objects and generate report.
  - Modify properties according to your workflow processes.
  - Access data from your in-house databases / systems and load such data onto S3D objects.
  - Perform Background scheduled tasks on Model / Catalog data for Object – like data analysis/checking/modification.
  - Create S3D objects in a non interactive environment.

Standalone Apps …

- Note that NO user interactive services/components (Client Services) can be used in such Standalone Apps. Those are available for use only in Interactive environment.
- You can only reference and use Middle Tier Services and Components.
- Whether you are running **S3D .net API** based Automation within Interactive TaskHost, or inside a Standalone Executable, the underlying semantics and associative framework is still running in the background. This "Central Processing Unit" of S3D keeps monitoring and reacts to modifications and percolates changes as per application specified metadata behavior.
- When you write Standalone App, you have to take care of
  - Connecting to a Site; (can connect to default last opened site, or provide site connection details)
  - Opening a Plant; (Can get a list of available Plants and their information)
  - Setting the Active Permission Group; (all modifications later on will be with this permission group)
  - Limitations: You cannot switch to a different plant once you have activated one plant.

Running the Standalone App

- For standalone exes to run, we must add the full path to **$Core\Runtime** and **$GeometryTopology\Runtime** folders to System Path. One can also do this using registry to setup custom paths for your exe name.
- Always use x86 configuration for the project. Will otherwise be a problem in 64 bit OS.
- Use only Single Threading Model.
- When Standalone EXEs are deployed and run, Runtime will not have a clue to pick up the CommonMiddle.dll from 3D Product Location. If you deploy a local copy of it, then you will not keep up with CommonMiddle.dll delivered by S3D product updates if any.
- One option to solve this is to deploy the Standalone EXE (and its dependencies) in Taskhost\Assemblies\Bin\Release\ directory and run it from there. This may be easy enough if your Standalone EXE doesn't have any more other dependencies to deploy.

Running the Standalone App …

- There is another technique to ensure that the CommonMiddle.dll assembly is loaded from the product location. This technique involves some additional code (to the Standalone exe) which basically provides runtime with a path to look for when it fails to load CommonMiddle.dll (we discuss that next).
- Once CommonMiddle.DLL this is is properly loaded, Middle Tier will properly load all other S3D .net API DLLs from the Product Location.

- This is not an issue in Interactive Taskhost running (because it ensures to load from Product location) and hence such code is NOT required for .net API based commands in Interactive Taskhost.

CommonMiddle.DLL path Resolution in Standalone EXEs

Add below call in Shared Constructor or other similar place where it will be executed first.

AddHandler AppDomain.CurrentDomain.AssemblyResolve, AddressOf MyResolveEventHandler

Add below function in form.

```
Shared Function MyResolveEventHandler(ByVal sender As Object, ByVal args As ResolveEventArgs) As [Assembly]
    'This handler is called only when the common language runtime tries to bind to the assembly and fails.
    'Retrieve the list of referenced assemblies in an array of AssemblyName.
    Dim objExecutingAssemblies As [Assembly] = [Assembly].GetExecutingAssembly
    Dim arrReferencedAssmbNames() As AssemblyName = objExecutingAssemblies.GetReferencedAssemblies()

    'Loop through the array of referenced assembly names.
    For Each strAssmbName As AssemblyName In arrReferencedAssmbNames
      'Look for the assembly names that have raised the "AssemblyResolve" event.
      If (strAssmbName.Name.EndsWith("CommonMiddle") AndAlso _
      strAssmbName.FullName.Substring(0, strAssmbName.FullName.IndexOf(",")) = args.Name.Substring(0, args.Name.IndexOf(","))) Then

        'We only have this handler to deal with loading of CommonMiddle. Rest everything we dont bother.
        RemoveHandler AppDomain.CurrentDomain.AssemblyResolve, AddressOf MyResolveEventHandler

        'Build the path of the assembly from where it has to be loaded.
        'Check CurrentUser,LocalMachine, 64bit CurrentUser,LocalMachine
        Const ProductInstallationKey As String = "\Software\Intergraph\SP3D\Installation"
        Const Wow6432Path = "\Software\Wow6432Node\Intergraph\SP3D\Installation"
        Const InstallDirKey = "INSTALLDIR"
        Dim sInstallPath = CStr(Registry.GetValue(Registry.CurrentUser.ToString & ProductInstallationKey, InstallDirKey, ""))

        If sInstallPath = "" Then  'try Local Machine
          sInstallPath = CStr(Registry.GetValue(Registry.LocalMachine.ToString & ProductInstallationKey, InstallDirKey, ""))
        End If

        If sInstallPath = "" Then   'try 64-bit CurrentUser
          sInstallPath = CStr(Registry.GetValue(Registry.CurrentUser.ToString & Wow6432Path, InstallDirKey, ""))
        End If

        If sInstallPath = "" Then    'try 64-bit LocalMachine
          sInstallPath = CStr(Registry.GetValue(Registry.LocalMachine.ToString & Wow6432Path, InstallDirKey, ""))
        End If

        If (Trim(sInstallPath) <> "") Then
          If (sInstallPath.EndsWith("\") = False) Then sInstallPath += "\"
          sInstallPath += "Core\Container\Bin\Assemblies\Release\"
        Else
          Throw New Exception("Error Reading SmartPlant 3D Installation Directory from Registry !!! Exiting")
        End If

        'Load the assembly from the specified path and return it.
        Dim strTempAssmbPath As String = sInstallPath & args.Name.Substring(0, args.Name.IndexOf(",")) & ".dll"
        Return [Assembly].LoadFrom(strTempAssmbPath)
      End If
    Next
    Return Nothing ' shouldnt reach here.
  End Function
```

S3D .net API

LAB

-

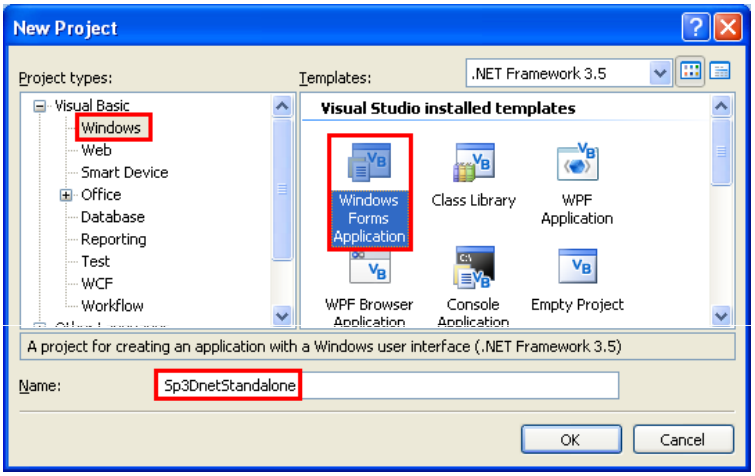
Writing Standalone Apps

LAB - Overview

- In this LAB we will learn
  - Writing Standalone Apps in VB.net
    - Creating Project, manage its settings
    - Adding required references and Importing required Namespaces
    - Adding the required functionality
      - Connect to Site
      - Activate Plant
      - Create a Property + ObjectType Filter
      - Create a Hierarchy + ObjectType Filter
      - Set Property using Generic Access
  - Running the Standalone App

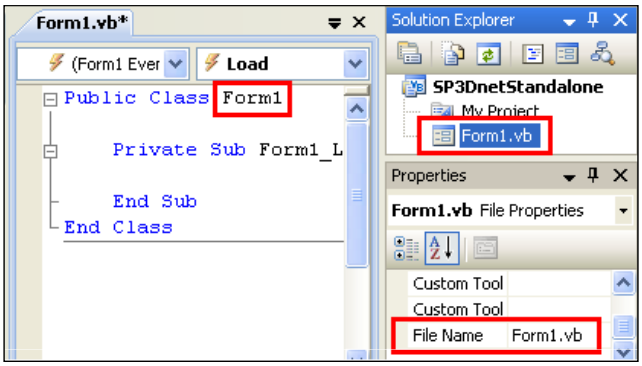
Create a VB.net Windows Form Application Project

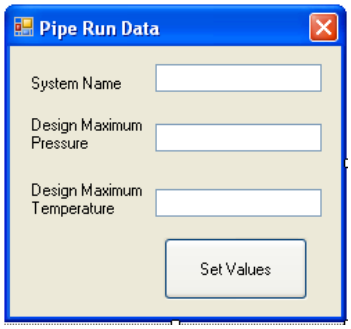
- Use Microsoft **Visual Studio 2008**.(Choose a .net language → VB.net / C#)
- Choose **File** > **New Project**
- Use **Visual Basic** > **Windows** from Project Types
- Pick **Windows Forms Application** template
- Specify **Name** of Project
- Hit **OK**.
- Project gets created.
- …



Create a VB.net Windows Form Application Project

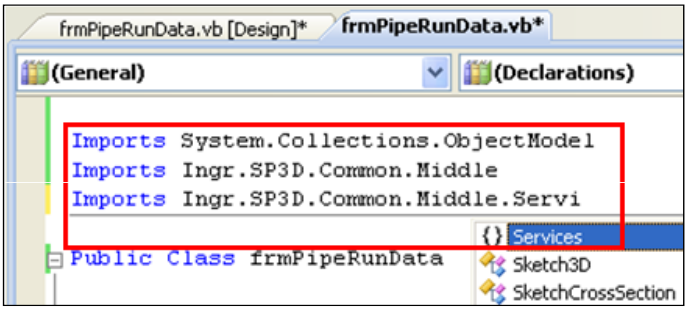
- It creates a Form1.vb which has an empty class named Form1.
- Rename the Name of the class from Form1 to a name of your choice and also rename the filename accordingly.
- Lets say we change it to "frmPipeRunData".
- Add Textbox and Button controls to the form.

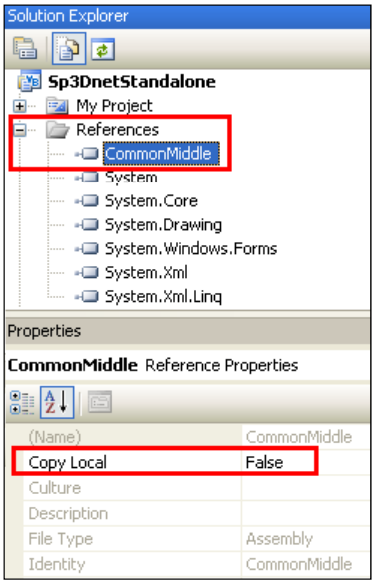




Adding required References, Import Required Namespaces

- Add the assembly “CommonMiddle.dll” as reference.
- Add the required “Imports” statements at the top of the file to enable you use the types in that namespace.





Adding the required functionality

```
Private Sub frmPipeRunData_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBa
    Try
        'Connect to Default Site. It is also possible to specify site connection info
        'and connect to it
        Dim oSite As Site
        oSite = MiddleServiceProvider.SiteMgr.ConnectSite()
        'to connect to a different than the default site use this:
        'oSite = MiddleServiceProvider.SiteMgr.ConnectSite("SOLIDS1002", "SP3DTrain_SDB", SiteManager.eD

        If oSite Is Nothing Then
            MsgBox("No Site Available")
        Else
            If oSite.Plants.Count > 0 Then
                oSite.OpenPlant(oSite.Plants(0)) 'choose the first
                'set permission group
                Dim oModel As Model = oSite.ActivePlant.PlantModel
                oModel.ActivePermissionGroup = oModel.PermissionGroups(0)
                MiddleServiceProvider.TransactionMgr.Commit("")
            Else
                MsgBox("No plants defined in Default Site")
            End If
        End If

    Catch ex As Exception
        MsgBox(ex.Message)
    End Try
End Sub
```

Can also Connect to a Site by providing Site Connection details

Not necessary to do this in a Custom Command. Site already active.

Can also open a Plant by Name
Can also get list of Available Plants, show them in a ComboBox and let the user Pick one.

Not necessary to do this in a Custom Command. Plant already active.

Note : To keep this lab simple, we open the first plant in the available plants list. For production use, you want to show list of Available Plants, let the user select one, call OpenPlant on that selected plant, list the available Permission Groups (PGs) in that plant, let user select a PG ... and then you are all set to do the rest of the logic.

Adding the required functionality

```
Private Sub btnSetValues_Click( . . .)
    Try
        ' First step :  Get all the systems with the given name using a Property Filter.
        Dim oPropertyFilter As New Filter, oProperty As PropertyValueString
        Dim oSystemsByName As ReadOnlyCollection(Of BusinessObject)
        oProperty = New PropertyValueString("IJNamedItem", "Name", txtSystemName.Text)
        oPropertyFilter.Definition.AddWhereProperty(oProperty, PropertyComparisonOperators.LIKE)
        oPropertyFilter.Definition.AddObjectType("Systems")
        oSystemsByName = oPropertyFilter.Apply() ' Apply the filter to get all the sytems with the given name.

        ' Second Step : Get all the PipeRuns under the Filtered objects that we get in First step.
        Dim oRunsFilter As New Filter, oRuns As ReadOnlyCollection(Of BusinessObject)
        oRunsFilter.Definition.AddHierarchy(HierarchyTypes.System, oSystemsByName, True)
        oRunsFilter.Definition.AddObjectType("Piping\PipingRuns")
        oRuns = oRunsFilter.Apply()
        If (oRuns.Count > 0) Then   ' Go through each run and set the required values.
            Dim dDsgMaxTemp As Double, dDsgMaxPressure As Double,  oRun As BusinessObject
            ' Get the given Temperature and Pressure values in model units.
            dDsgMaxPressure = m_oUOM.ParseUnit(UnitType.ForcePerArea, txtDsgMaxPressure.Text)
            dDsgMaxTemp = m_oUOM.ParseUnit(UnitType.Temperature, txtDsgMaxTemp.Text)
            For Each oRun In oRuns
                oRun.SetPropertyValue(dDsgMaxTemp, "IJProcessDataInfo", "DesignMaxTemp")
                oRun.SetPropertyValue(dDsgMaxPressure, "IJProcessDataInfo", "DesignMaxPressure")
                MiddleServiceProvider.TransactionMgr.Commit("Set Value")
            Next
        Else
            MsgBox("No pipe runs found.")
        End If
    Catch ex As Exception
        MsgBox(ex.Message)
    End Try
End Sub
```


Ending Standalone Apps

- Add the below to the logic which handles ending of a Standalone EXE
  - MiddleServiceProvider.TransactionMgr.Abort
  - MiddleServiceProvider.Cleanup

```
Private Sub frmPipeRunData_FormClosing(ByVal sender As Obje
  MiddleServiceProvider.Transactionmgr.Abort()
  MiddleServiceProvider.cleanup()
End Sub
```

CommonMiddle.DLL path Resolution in Standalone EXEs

Add below call in a Shared Constructor (Shared Sub New) of the Form, or Main class.

```
AddHandler AppDomain.CurrentDomain.AssemblyResolve,
                AddressOf MyResolveEventHandler
```

Add below function in the same form/class.

```
Shared Function MyResolveEventHandler(ByVal sender As Object, ByVal args As
        ResolveEventArgs) As [Assembly]
  'This handler is called only when the common language runtime tries to bind to the
        assembly and fails.
  'Retrieve the list of referenced assemblies in an array of AssemblyName.
  Dim objExecutingAssemblies As [Assembly] = [Assembly].GetExecutingAssembly
  Dim arrReferencedAssmbNames() As AssemblyName =
        objExecutingAssemblies.GetReferencedAssemblies()

  'Loop through the array of referenced assembly names.
  For Each strAssmbName As AssemblyName In arrReferencedAssmbNames
    'Look for the assembly names that have raised the "AssemblyResolve" event.
    If (strAssmbName.Name.EndsWith("CommonMiddle") AndAlso _
    strAssmbName.FullName.Substring(0, strAssmbName.FullName.IndexOf(",")) =
        args.Name.Substring(0, args.Name.IndexOf(","))) Then

    'We only have this handler to deal with loading of CommonMiddle. Rest everything
        we dont bother.
    RemoveHandler AppDomain.CurrentDomain.AssemblyResolve, AddressOf
        MyResolveEventHandler

    'Build the path of the assembly from where it has to be loaded.
    'Check CurrentUser,LocalMachine, 64bit CurrentUser,LocalMachine
    Const ProductInstallationKey As String = "\Software\Intergraph\SP3D\Installation"
    Const Wow6432Path = "\Software\Wow6432Node\Intergraph\SP3D\Installation"
    Const InstallDirKey = "INSTALLDIR"
```

Contd…

Contd …

```
    ' Find info in registry from HKCU, HKLM 32 bit and then HKCU, HKLM 64bit.
    Dim sInstallPath = CStr(Registry.GetValue(Registry.CurrentUser.ToString &
        ProductInstallationKey, InstallDirKey, ""))

    If sInstallPath = "" Then  'try Local Machine
      sInstallPath = CStr(Registry.GetValue(Registry.LocalMachine.ToString &
        ProductInstallationKey, InstallDirKey, ""))
    End If

    If sInstallPath = "" Then   'try 64-bit CurrentUser
      sInstallPath = CStr(Registry.GetValue(Registry.CurrentUser.ToString &
        Wow6432Path, InstallDirKey, ""))
    End If

    If sInstallPath = "" Then    'try 64-bit LocalMachine
      sInstallPath = CStr(Registry.GetValue(Registry.LocalMachine.ToString &
        Wow6432Path, InstallDirKey, ""))
    End If

    If (Trim(sInstallPath) <> "") Then
      If (sInstallPath.EndsWith("\") = False) Then sInstallPath += "\"
      sInstallPath += "Core\Container\Bin\Assemblies\Release\"
    Else
      Throw New Exception("Error Reading SmartPlant 3D Installation
        Directory from Registry !!! Exiting")
    End If

    'Load the assembly from the specified path and return it.
    Dim strTempAssmbPath As String = sInstallPath & args.Name.Substring(0,
        args.Name.IndexOf(",")) & ".dll"
    Return [Assembly].LoadFrom(strTempAssmbPath)
    End If
  Next
  Return Nothing ' shouldnt reach here.
  End Function
```


Running / Debug the Standalone App Project

- Save the project to a location on the disk and build the project.
- (as discussed earlier) Remember to add $Core\Runtime and $GeometryTopology\Runtime to System Path . This is required for Standalone EXEs to run.
- (as discussed earlier) Ensure you have taken care of getting CommonMiddle.DLL loaded correctly with one of the options we learnt earlier.
- If running it inside Developer Studio, Run / Debug the project.
- If running it directly as EXE, just start the exe.

S3D .net API

Application Functionality

–

Systems & Specs

Application Functionality – Systems & Specs

Assembly : SystemMiddle.dll

Constructors : All of same syntax like

AreaSystem (oParentSystem)

Interfaces : All Systems implement below

ISystem → System Parent Behavior

SystemChildren collection

AddSystemChild method

ISystemChild → SystemChild Behavior

Get/Set SystemParent property

IAllowableSpecs → Allowed Specs Management

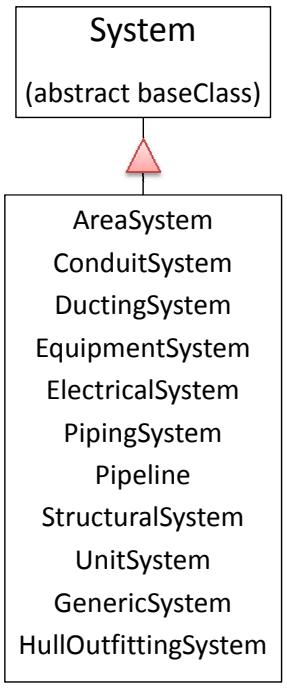
Get/Set/Add/Remove/Replace/Reset Allowed Specs.

INamedItem, INameRule → NamedItem & NameRule behavior,

Get Name, Get/Set ActiveNameRule, SetUserDefinedName

Properties : Get/Set using Generic Property Access

Types of Systems / Class Hierarchy.



S3D .net API

Application Functionality

–

Equipment

Application Functionality – Equipment

Assembly : EquipmentMiddle.dll

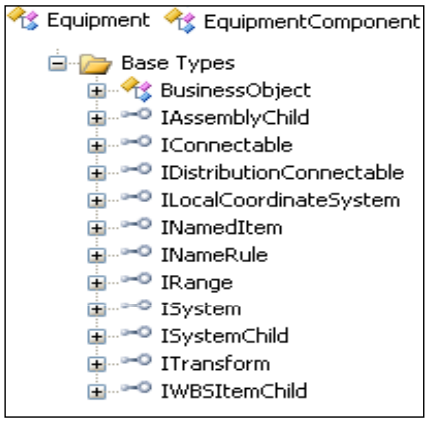
Classes :

Equipment, EquipmentComponent, Shape
PipeNozzle, FoundationPort, CableTrayPort, ConduitPort, CablePort, HVACPort

Constructors : All of same syntax like

Equipment (oPart As IPart, oParentSystem)
Equipment (oPartCls As IPartClass, oParentSystem)
Equipment (strPartNo As String, oParentSystem)
EquipmentComponent (oPart As IPart, oParentEqp)
EquipmentComponent (oPartCls As IPartClass, oParentEqp)
EquipmentComponent(strPartNo As String, oParentEqp)
Shape (strPartNo, oParent)
Shape (oPart, oParent)

Class Hierarchy, Implemented Interfaces



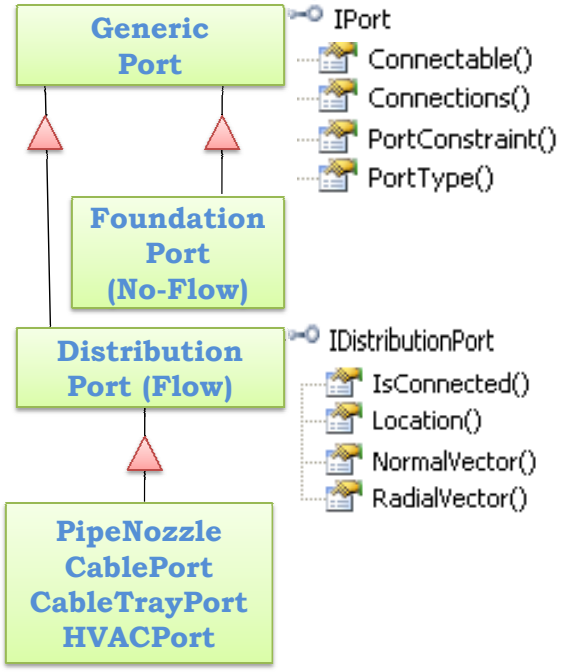
Shape → Same as above except
IAssemblyChild, IConnectable, IDistributionConnectable, ISystem

Properties :

Equipment / EquipmentComponent / Shape → Origin, Matrix, Range, Constraints, XYZ Axis.
Shape → Representation, Part
For all other Properties use Generic Property Access

Application Functionality – Equipment – Ports/Nozzles

Class Hierarchy, Implemented Interfaces Available Properties (Important ones)



Constructors :

Distribution Port Constructors : All Distribution Ports offer constructors with same syntax as below.

- **PipeNozzle** (strPartNo, bLightWtGraphics, PortIndex, oParentSystem)
- **PipeNozzle** (strPartNo, Connection, bLightWtGraphics, PortIndex)
- **PipeNozzle** (strPartNo, Connection, bLightWtGraphics, PortIndex, oPosition, oNormal, dLength, bIsFacePosition)

Foundation Port Constructors.

- **FoundationPort** (strPartNo, PortIndex, oEqp)
- **FoundationPort** (strPartNo, PortIndex, oEqpComp)

Properties : PipeNozzle, HVACNozzle → Length.

NozzleOrientation → PortConstraint

S3D .net API

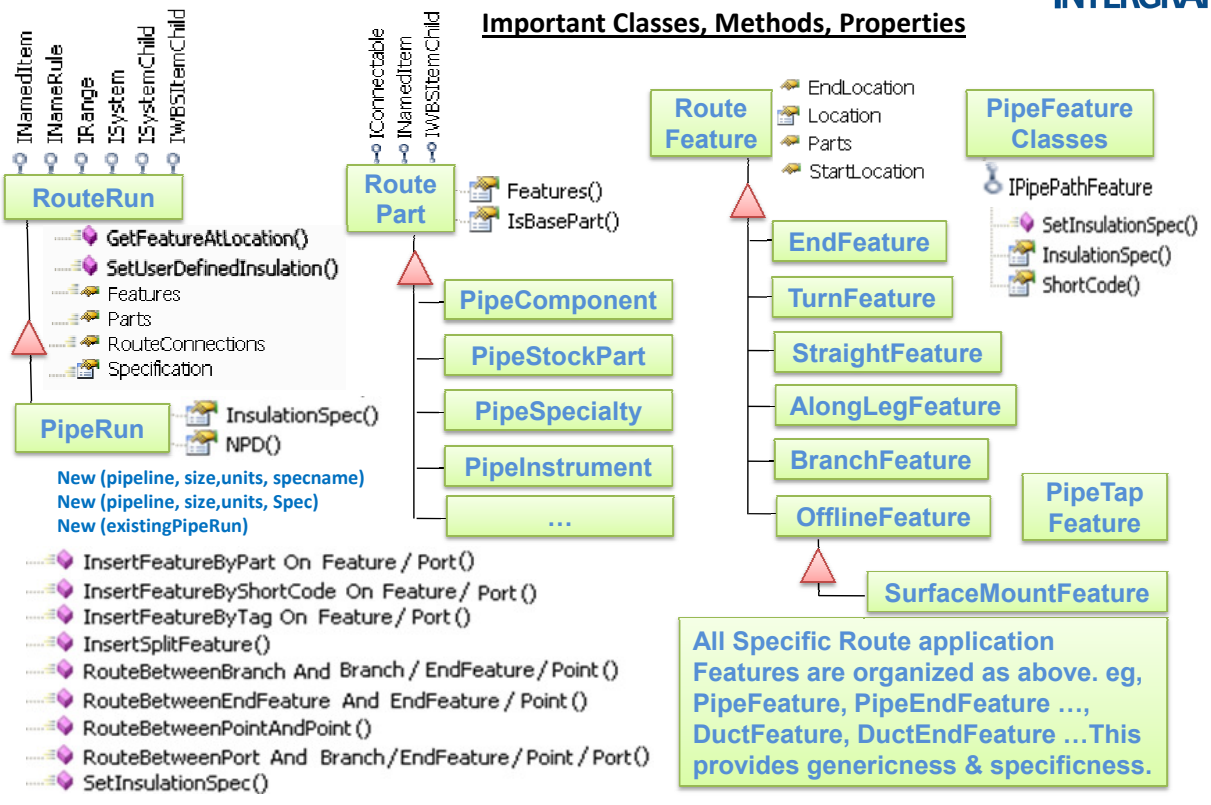
Application Functionality

–

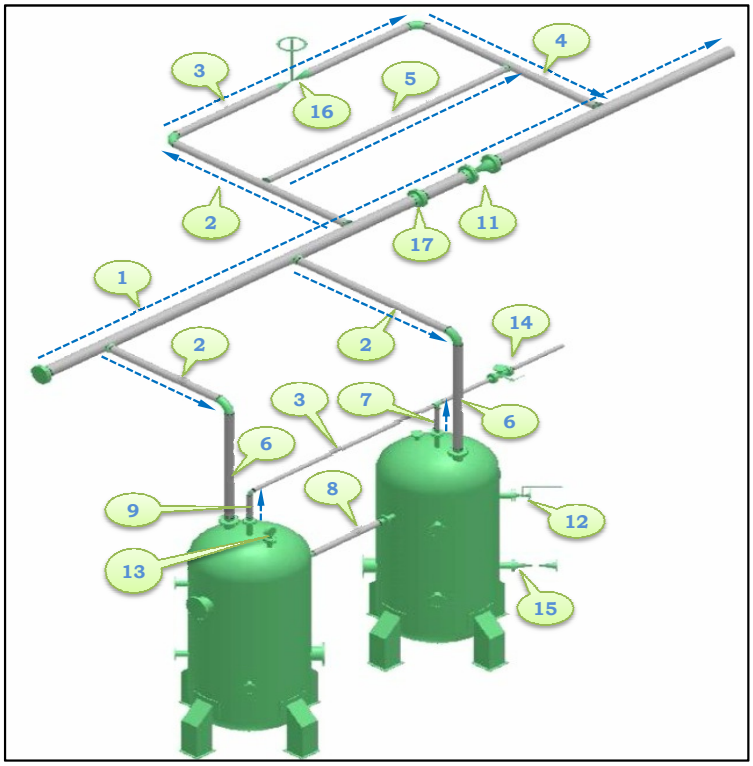
Route

Application Functionality – Route

Important Classes, Methods, Properties



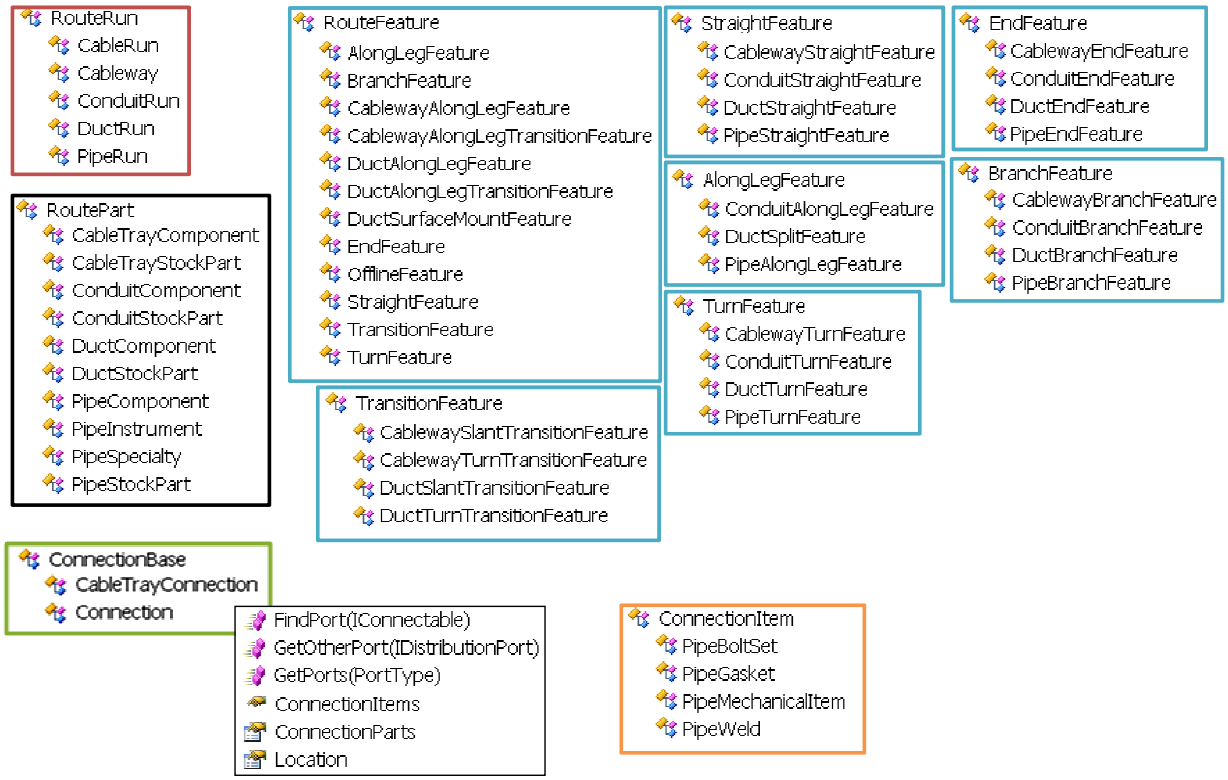
Usage of Various Routing Methods.



RouteBetween variants
- 01 – PointAndPoint ( )
- 02 – BranchAndPoint ( )
- 03 – EndFeatureAndPoint ( )
- 04 – EndFeatureToBranch ( )
- 05 – BranchAndBranch ( )
- 06 – PortAndEndFeature ( )
- 07 – PortToBranch ( )
- 08 – PortToPort ( )
- 09 – PortAndPoint ( )

InsertFeature variants
- 11 – ByShortCodeOnFeature ( )
- 12 – ByShortCodeOnPort ( )
- 13 – ByTagOnPort ( )
- 14 – ByTagOnFeature ( )
- 15 – ByPartOnPort ( )
- 16 – ByPartOnFeature ( )
- 17 – InsertSplitFeature ( )

Route – Object Model



Other Discipline Routing Methods

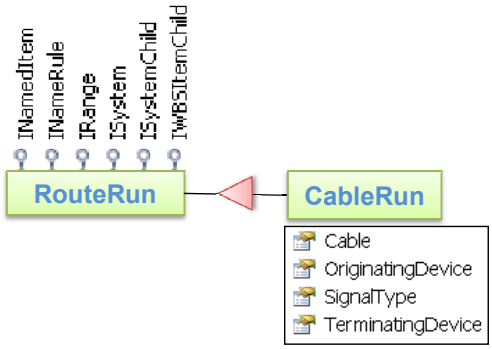
Conduit :

- Same concept as Piping for Routing, Inserting Components, connecting Branches etc.
- Slight Terminology change. Uses NCD (Nominal Conduit Diameter) instead of NPD.
- No Specialties / Instruments, Insulation.

Cableway / Ducting:

- Same concept as Piping for Routing, Inserting Components, connecting Branches etc.
- Additional Parameter of CrossSection, and its orientation.
- No Specialties / Instruments, and No Insulation for Cableway.

Cable Runs:



New (ISystem parentSystem, IPart CablePart,
IConnectable OriginatingDevice, IConnectable TerminatingDevice)
ConnectToPortOnOriginatingDevice (IPort oElectricalPort)
ConnectToPortOnTerminatingDevice (IPort oElectricalPort)

S3D .net API

Demo / Code Walkthrough

–

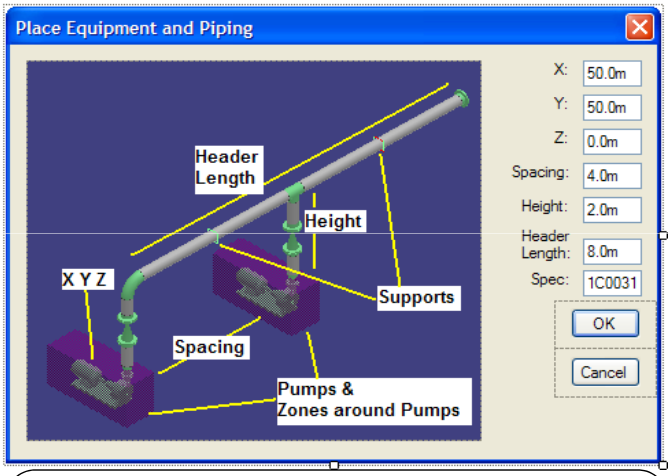
Overall Application Command

–

Systems, Equipment, Route, Spaces, Hangers, Access Catalog …

Demo / Code walkthrough

- Demonstration and walkthrough of a Sample involving application objects.
- We will review below in this Custom Command
  - Create Generic System, PipelineSystem
  - Set Attributes (Double, Codelisted, String)
  - Create Notes on Objects
  - Create Space Folders, Spaces
  - Associate objects to Spaces
  - Access Nozzles on Equipment
  - Create PipeRuns, Route PipeRuns,
  - Using Routing API
  - Routing - Branching, Insert Component
  - Finding Features at a Location.
  - Math API – Points, Vector, Matrix math.
  - Access Part in Catalog given PartNumber.
  - Add Design Support on Pipe.



The modeling configuration this Sample Command produces is shown here. It is to demonstrate the Automation Capability and Simplicity and give a broad idea of current coverage level.

Implementation of Command & Form

Implementation of Command

Just inherit from BaseModalCommand and override OnStart ( ) method

```
Imports Ingr.SP3D.Common.Client

Public Class PlaceEqpAndPiping
    Inherits BaseModalCommand 'Ours is a simple Modal Command with a Form.

    Public Overloads Overrides Sub OnStart(. . .)
        ' Create and show our form
        Dim myForm As New PlaceEqpAndPipingForm
        myForm.ShowDialog()
    End Sub

End Class
```

Implementation of Form

add **class variables** to hold parameters keyed in by user which placement will use.

On **Form Load** – assign initial defaults to parameter variables as shown on the form.

```
Public Class PlaceEqpAndPipingForm
  ' Define Parameters used for Placement.
  Private m_dSpacing As Double ' -> Spacing between pumps.
  Private m_dHeaderLength As Double ' -> Length of Header
  Private m_dHeight As Double ' -> Height Of Header above Pump Discharge.
  Private m_strSpec As String ' -> The Spec to Use
  Private m_dX, m_dY, m_dZ As Double ' -> The Location of the 1st Pump

  Private Sub PlaceEqpAndPipingForm_Load(. . .) Handles MyBase.Load
    ' Initialize the values to the default values on the form.
    m_dX = 50
    m_dY = 50
    m_dZ = 0
    m_dSpacing = 4
    m_dHeaderLength = 8
    m_dHeight = 2
    m_strSpec = "1C0031"
  End Sub
```


Form Implementation –Validate & Gather inputs, Process Actions

```
Private Sub txtX_KeyDown(. . .) Handles txtX.KeyDown
  If (e.KeyCode = Keys.Enter) Then ParseDistance(txtX.Text, txtX, m_dX)
End Sub

Private Sub txtX_LostFocus(. . .) Handles txtX.LostFocus
  ParseDistance(txtX.Text, txtX, m_dX)
End Sub
```

Implementation of **KeyDown & LostFocus**

For all Distance fields - X,Y,Z, Spacing, HeaderLength, Height

On Enter Key – call ParseDistance – if valid it store values in parameter class variables, if invalid it shows the field in red.

For Spec field, it stores into m_strSpec parameter.

```
Private Sub Cancel_Button_Click(. . .)
  Me.DialogResult = DialogResult.Cancel
  ClientServiceProvider.TransactionMgr.Abort()
  Me.Close()
End Sub
```

On Cancel Button Press, just abort transaction and close form.

```
' This method parses distance keyin and returns distance
' visually indicates if it is in error.
Private Function ParseDistance(ByVal txt As String, ByVal ctrl As TextBox, ByRef dValue As Double) As Double
  Dim dVal As Double
  Try
    dVal = MiddleServiceProvider.UOMMgr.ParseUnit(UnitType.Distance, txt)
    dValue = dVal
    ctrl.ForeColor = Drawing.Color.Black
  Catch e As Exception
    ctrl.ForeColor = Drawing.Color.Red
  End Try
End Function
```

Function to Parse Distance keyed in by user. Indicates (in red) if keyin value is invalid; returns the parsed value if the keyin value is valid.

Main Functionality – Creating Objects

On **OK** button press

Get Active Plant's Model.

Get RootSystem from the Model.

Create a Generic System under RootSystem, and Set its Name using the SetUserDefinedName method on it.

Commit the transaction.

```
Private Sub OK_Button_Click(. . .) Handles OK_Button.Click
  Me.DialogResult = System.Windows.Forms.DialogResult.OK

  ' Get the active plant's Model.
  Dim oModel As Model
  oModel = MiddleServiceProvider.SiteMgr.ActiveSite.ActivePlant.PlantModel

  ' Get Plant's Root System
  Dim oRootSys As BusinessObject
  oRootSys = oModel.RootSystem

  ' Create a Generic System under Plant Root System
  Dim oGenericSys As New GenericSystem(oRootSys)
  oGenericSys.SetUserDefinedName("New Generic System")
  ClientServiceProvider.TransactionMgr.Commit("Create New Generic System")
```

Create a New Pipeline under the Generic System

Set Name, SequenceNumber and FluidCode on PipeLine.

Use MetadataMgr to resolve CodelistText to number.

Commit changes.

```
'Create a new Pipeline under the newly created Generic System, and set its Properties
Dim oPipeLine As Pipeline = New Pipeline(oGenericSys)
oPipeLine.SetUserDefinedName("New Pipeline")
oPipeLine.SetPropertyValue("0101", "IJPipelineSystem", "SequenceNumber")

' Let's set FluidCode to 'OH' --> Hydraulic Oil
Dim oCLI As CodelistItem
oCLI = oModel.MetadataMgr.GetCodelistInfo("FluidCode", "CMNSCH").GetCodelistItem("OH")
oPipeLine.SetPropertyValue(oCLI, "IJPipelineSystem", "FluidCode")

' Another way to set FluidCode to "OH" is to use its
' Integer value (=503, see MetadataBrowser), as done below
'oPipeLine.SetPropertyValue(503, "IJPipelineSystem", "FluidCode")

' The below statement will save changes since previous save to database.
' The created Pipeline will be saved to database.
ClientServiceProvider.TransactionMgr.Commit("Create New Pipeline")
```


Main Functionality – Place Pumps & Add Notes.

Create the main Pump using Equipment Constructor, giving part number & parent system.

Set its Origin & Orientation.

```
' Create TWO pumps, The first one created here
Dim oPump1 As Equipment, oPump2 As Equipment, oMtx As New Matrix4X4
oPump1 = New Equipment("HCPump05 8""x4""-E", oGenericSys)
oMtx.SetIdentity()
oPump1.Matrix = oMtx ' --> Control Orientation with Matrix
oPump1.Origin = New Position(m_dX, m_dY, m_dZ) '--> Position specified on Form
```

Create & Add Notes to the Pump. Use Note Constructor passing Parent (pump). Set Properties using Generic Property access

```
'Create and attach a Note to the Pump created above.
Dim oNote As Note
oNote = New Note(oPump1) '--> Create Note on Pump1
oNote.SetPropertyValue(5, "IJGeneralNote", "Purpose") ' Maintenance
oNote.SetPropertyValue("This is MAIN Pump", "IJGeneralNote", "Text")
oNote.SetPropertyValue("Operation Information", "IJGeneralNote", "Name")
oNote = New Note(oPump1) '--> Create Another Note on Pump1
oNote.SetPropertyValue(6, "IJGeneralNote", "Purpose") ' Inspection
oNote.SetPropertyValue("Inspect Every Month", "IJGeneralNote", "Text")
oNote.SetPropertyValue("Inspection Information", "IJGeneralNote", "Name")
```

Create the backup Pump at required *spacing* from main pump and add Notes. Same as above – note text modified as appropriate.

```
'Create another pump, our Backup pump.
oPump2 = New Equipment("HCPump05 8""x4""-E", oGenericSys)
oPump2.Matrix = oMtx
oPump2.Origin = New Position(m_dX + m_dSpacing, m_dY, m_dZ)
```

```
oNote = New Note(oPump2) '--> Create Note on Pump2
oNote.SetPropertyValue(5, "IJGeneralNote", "Purpose") ' Maintenance
oNote.SetPropertyValue("This is STANDBY/BACKUP Pump", "IJGeneralNote", "Text")
oNote.SetPropertyValue("Operation Information", "IJGeneralNote", "Name")
oNote = New Note(oPump2) '--> Create Another Note on Pump2
oNote.SetPropertyValue(6, "IJGeneralNote", "Purpose") ' Inspection
oNote.SetPropertyValue("Inspect Every 3 Months", "IJGeneralNote", "Text")
oNote.SetPropertyValue("Inspection Information", "IJGeneralNote", "Name")
```

Compute & Commit changes.

Pumps, Notes get saved and show in graphics now.

```
' Compute shows Dynamics (all modified objects in rubberband hilite.). Commit saves
' the objects (two pumps, notes, with relation to pumps) and modifications to database.
ClientServiceProvider.TransactionMgr.Compute()
ClientServiceProvider.TransactionMgr.Commit("Place Two Pumps")
```

Main Functionality – Set NameRule, Create SpaceFolder, Spaces

Set Pumps to use **UniqueNameRule**.

CatalogBaseHelper provides available NameRules given the object type.

Find UniqueNameRule among them and Set **ActiveNameRule** property.

```
'Lets find available Name Rules for Equipment class and pick 'UniqueNameRule' if it exists.
' For this, we use CatalogHelper to find out available Name rules on an object Class type.
' See GenericNameRules sheet.
Dim oCatalogHelper As New CatalogBaseHelper, oUniqNameRule As BusinessObject = Nothing
For Each oNameRule As BusinessObject In oCatalogHelper.GetNamingRules("CPEquipment")
  'If we got our name rule of interest, then we exit this loop
  Dim oPVS As PropertyValueString = oNameRule.GetPropertyValue("IJDNameRuleHolder", "Name")
  If oPVS.PropValue = "UniqueNameRule" Then
    oUniqNameRule = oNameRule
    Exit For
  End If
Next oNameRule
If (oUniqNameRule IsNot Nothing) Then
  oPump1.ActiveNameRule = oUniqNameRule
  oPump2.ActiveNameRule = oUniqNameRule
End If

' The compute here evaluates the changes since last compute/commit i.e. the name rule assignment.
' Here the Name Rule runs, generates and assigns a name.
ClientServiceProvider.TransactionMgr.Compute()
' Commit saves the changes i.e. name rule assignment to the two pumps.
ClientServiceProvider.TransactionMgr.Commit("Assign Unique NameRule to Pumps")
```

Create a Space Folder under root and ZONE Spaces by Two points around the pump range volume.

The utility functions used here are explained later

```
' Create a SpaceFolder, in which we will create ZoneSpaces around the Pumps
Dim oSpaceFolder As FolderSpace
oSpaceFolder = CreateSpaceFolder("Space Folder for Spaces around Pumps")
ClientServiceProvider.TransactionMgr.Commit("Create Pumps Space Folder")

' Now lets create ZoneSpaces
CreateSpace(oPump1, oSpaceFolder, "ZONETYPE", SpaceConstructionType.SpaceBy2Points)
CreateSpace(oPump2, oSpaceFolder, "ZONETYPE", SpaceConstructionType.SpaceBy2Points)
ClientServiceProvider.TransactionMgr.Commit("Create Spaces for Pumps")
```


Main Functionality – Create Runs, Route out of Pump Nozzle

Create 1st PipeRun with Constructor Passing PipeLine, Size & Spec.

Set Process Data, Flow Direction using Generic Property Access and Set Name [UserDefinedName]

Compute changes done so far.

Now, for the 2nd run, use Copy constructor passing 1st Run. All properties applied from source run. (Features / Routing not copied.)

```
' Route out from Pump1's Nozzle #1
' Create a PipeRun, 8in, of given spec
Dim oPipeRun1 As PipeRun = New PipeRun(oPipeLine, 8.0, "in", m_strSpec)

' Set Temperatures and Pressures on PipeRun and Flow Direction
' The SetPropertyValue has several overloads to deal with different attribute types
oPipeRun1.SetPropertyValue(500.0, "IJProcessDataInfo", "DesignMaxTemp")
oPipeRun1.SetPropertyValue(7500.0, "IJProcessDataInfo", "DesignMaxPressure")
oPipeRun1.SetUserDefinedName("Run - 01") ' -> Set Name on PipeRun
' Set Flow Direction. Could have also done the CodelistItem way
oPipeRun1.SetPropertyValue(1, "IJRtePipeRun", "FlowDirection")
ClientServiceProvider.TransactionMgr.Compute()
' -> Note, Run is still temporary, not yet saved to Database.

' Clone another run with same properties, for the 2nd Pump.
Dim oPipeRun2 As PipeRun
oPipeRun2 = New PipeRun(oPipeRun1) '-> Create a new Run from an existing run
oPipeRun2.SetUserDefinedName("Run - 02")
```

We get the Pump Nozzle using GetPortByName utility function (explained later) and then route out of it. We get its Location and then construct the next point using Point/Vector Math API and then use Routing methods to route.

```
' Route dHeight distance outward from Pump1 Discharge
'   Get the Discharge port of Pump1
Dim oDischargeOfPump1 As IDistributionPort
oDischargeOfPump1 = GetPortByName(oPump1, "Discharge")

Dim oPos1 As Position, oFeat As IRoutePathFeature
' Calculate the Point at a distance of dHeight from the Nozzle Location
' to do this, we Initialize a vector with appropriate values, and
' use it to offset the Nozzle Location Point
oPos1 = oDischargeOfPump1.Location.Offset(New Vector(0, 0, m_dHeight))

'Route from Pump Discharge to that point dHeight above.
'Below method returns the resulting end feature for route continuation.
oPipeRun1.RouteBetweenPortAndPoint(oDischargeOfPump1, oPos1, oFeat)
```

RouteBetweenPortAndPoint returns the ***End Feature***, needed for *further routing …*

Main Functionality – Continue Routing

We compute here to see the routed pipe in dynamics. Solvers kick in and generate flanges, connections, reducers, welds, bolts, gaskets as needed. All of these parts are picked up from catalog as per spec rules. This Rule based system allows you to concentrate on your automation logic and system reacts to do necessary ground work.

```
' The below Compute statement triggers evaluation logic in SP3D for the
' changes done so far to objects. It will generate
'  Connection to nozzle with Mating Flange, bolts & gaskets, reducer if required,
'  weld for the Reducer, plus the pipe upto the location specified.
'  all of the parts/connection items will be picked up based on the spec and its rules.
ClientServiceProvider.TransactionMgr.Compute()
```

```
' Route 'dHeaderLength' in East Direction from Current End Position.
' We offset the current point with a vector (direction & length) to move.
Dim oPos2 As Position
oPos1 = New Position(oFeat.Location)
oPos2 = oPos1.Offset(New Vector(m_dHeaderLength, 0, 0))
' Now we route from the EndFeature to that Point
oPipeRun1.RouteBetweenEndFeatureAndPoint(oFeat, oPos2, oFeat)

' In the Compute Call below, the elbow plus the pipe and welds get generated.
ClientServiceProvider.TransactionMgr.Compute()
```

RouteBetweenEndFeatureAndPoint returns the ***End Feature***, needed for *further routing* to a next *Point / Branch Header / Port …*

We finally insert a flange, compute and then commit. The entire run gets saved to database.

```
' We now place a terminal Flange at the end of this pipe. The oFeat we have with
' us from above call is the current end feature till the last routed point.
Dim oPF As RouteFeature
oPipeRun1.InsertFeatureByShortCodeOnFeature("Flange", 1, "", oFeat, oFeat.Location, Nothing, oPF)
' In the Compute Call below, the inserted feature is evaluted for placement as per the spec rules.
ClientServiceProvider.TransactionMgr.Compute()

' The below statement will save changes since the previous save to database.
' i.e. the Mating Flanges, Bolts, Gaskets, Welds will be saved to database.
ClientServiceProvider.TransactionMgr.Commit("Place 1st Run")
```

InsertFeatureByShortCodeOnFeature places item on Feature (Straight / End) at given point.

Main Functionality – Continue Routing

```
'Now, lets Route out of 2nd Pump Discharge to branch in into the already placed header (run1)
Dim oDischargeOfPump2 As IDistributionPort
oDischargeOfPump2 = GetPortByName(oPump2, "Discharge")
'Arrive at the required position on Header, i.e. dHeight from the Nozzle Location
oPos1 = oDischargeOfPump2.Location.Offset(New Vector(0, 0, m_dHeight)) ' move dHeight in Z direction
' Below call is to
'  Locate the Feature representing the Straight Pipe
'  At position (oPos1) [could be approximate position within a tolerance]
'  Returns accurate branching point (oPos2) and the Feature (oFeat)
oPipeRun1.GetFeatureAtLocation(PathFeatureType_STRAIGHT, PathFeatureFunction_ROUTE, oPos1, oPos2, oFeat)

' Now, Branch in into the Header (Run1) from the Discharge Nozzle of Pump2.
oPipeRun2.RouteBetweenPortAndBranch(oDischargeOfPump2, oFeat, oPos2)
```

GetFeatureAtLocation returns Feature on a given Run, at a given Location. Needed for inserting components / branching in or out in a new run to a next *Point / Branch Header / Port / End Feature…*

We Compute & Commit here – in addition to earlier explained situation, System here solves to split Header Pipe into two, generate Branch Components on Header, and connects the branch and header runs – all driven by in-built intuitive modeling rules and configured spec rules.

```
' The below compute statement generates Mating Flange, Bolts
' Gaskets, Reducer (if required), welds, plus Tee on the
' header, and connects the Tee's 3rd port to Branch Run.
ClientServiceProvider.TransactionMgr.Compute()
' Save changes to database.
ClientServiceProvider.TransactionMgr.Commit("Place 2nd Run")
```

This 'Rule Based Reactive Processing' showcases the Automation Potential of Smart 3D – user automates his needs and the system reacts with inherent Metadata driven functionality – *a* ***key differentiator*** from competitor product technologies.

Main Functionality – Insert Components & Supports on Pipe

Here we place Check Valves on the Vertical Pipe legs connecting to pump nozzles, using **GetFeatureAtLocation** and **InsertFeatureByShortCodeOnFeature** methods.

```
' Now lets place Check Valves on the vertical Pipes connected to the Main and Standby Pumps
Dim oValveFeat As IRoutePathFeature
' Arrive at the Point where you want to place the valve.
oPos1 = oDischargeOfPump1.Location.Offset(New Vector(0, 0, 0.5 * m_dHeight))
' Get the Straight feature on the run at that location, we need it to insert the valve.
oPipeRun1.GetFeatureAtLocation(PathFeatureType_STRAIGHT, PathFeatureFunction_ROUTE, _
                               oPos1, oPos2, oFeat)
' Below method inserts a feature by "ShortCode & Option" on straight feature at a given position.
oPipeRun1.InsertFeatureByShortCodeOnFeature("Check Valve", 1, "", oFeat, oPos2, Nothing, oValveFeat)

' Now do the same on the 2nd vertical pipe connected to the standby Pump.
oPos1 = oDischargeOfPump2.Location.Offset(New Vector(0, 0, 0.5 * m_dHeight))
oPipeRun2.GetFeatureAtLocation(PathFeatureType_STRAIGHT, PathFeatureFunction_ROUTE, _
                               oPos1, oPos2, oFeat)
oPipeRun2.InsertFeatureByShortCodeOnFeature("Check Valve", 1, "", oFeat, oPos2, Nothing, oValveFeat)

' The below compute statement will place Check Valves, generate Mating Flanges,
' associated Bolts/Gaskets, welds ... Shows dynamics and gets ready for Commit.
ClientServiceProvider.TransactionMgr.Compute()
' Save changes to database.
ClientServiceProvider.TransactionMgr.Commit("Place Check Valves")
```

We place a DesignSupport, constructed with **Pipeline, Supported objects (pipes), Supporting objects (steel) & Position** as inputs and set the Support Part **(SupportDefinition)**

```
' Lets place supports on the two Horizontal Pipes of the main header.
' Get the position where we will look for the Straight Pipe.
oPos1 = oDischargeOfPump1.Location.Offset(New Vector(0.25 * m_dHeaderLength, 0, m_dHeight))
' Get the Straight Pipe
oPipeRun1.GetFeatureAtLocation(PathFeatureType_STRAIGHT, PathFeatureFunction_ROUTE, _
                               oPos1, oPos2, oFeat)
'Now we create a Support at that point. Prepare a Supported Objects Collection.
Dim oSupportedCol As New Collection(Of RouteFeature) '-> Supported Collection --> Feature(s)
oSupportedCol.Add(oFeat)
' Create the Support with required inputs - owning Pipeline, SupportedFeatures,
'   SupportingFeatures (structure - not applicable for design support we are using)
Dim oSupp As DesignSupport
oSupp = New DesignSupport(oPipeLine, oSupportedCol, Nothing, oFeat.Location)
' Specify the Designed Pipe Support Catalog Part to use for this Design Support.
oSupp.SupportDefinition = oCatalogHelper.GetPart("DesignPipeH_1")
```


Main Functionality – Place Supports…, GetPortByName

We place a DesignSupport on the 2nd Pipe too.

Compute and Commit changes and,

finally close the form.

```
  ' Now for the 2nd Pipe. Locate it by Going UP dHeight from Pump2's Discharge and
  ' then 25% of headerlength to the East
  oPos1 = oDischargeOfPump2.Location.Offset(New Vector(0.25 * m_dHeaderLength, 0, m_dHeight))
  oPipeRun1.GetFeatureAtLocation(PathFeatureType_STRAIGHT, PathFeatureFunction_ROUTE, _
                                 oPos1, oPos2, oFeat)
  'Clear existing SupportedFeature and add the 2nd Pipe alone.
  oSupportedCol.Clear()
  oSupportedCol.Add(oFeat)
  oSupp = New DesignSupport(oPipeLine, oSupportedCol, Nothing, oFeat.Location)
  oSupp.SupportDefinition = oCatalogHelper.GetPart("DesignPipeH_1")

  ' Compute - evaluates changes/ Objects created so far and shows dynamics.
  ' Supports associated to Pipe Features and Pipeline owns the supports.
  ClientServiceProvider.TransactionMgr.Compute()
  ' Save changes to database.
  ClientServiceProvider.TransactionMgr.Commit("Place Supports")

  MsgBox("Placement Complete", , "Place Equipment and Piping")
  Me.Close()
End Sub
```

```
' The below routine returns a Port by Name on a Part
Private Function GetPortByName(ByVal oPart As IConnectable, ByVal strName As String) As IPort
  Dim oPort As IPort
  GetPortByName = Nothing ' -> Initialize
  For Each oPort In oPart.GetPorts(PortType.All) ' Search in All Ports
    If oPort.ToString = strName Then ' For given portName
      GetPortByName = oPort ' Return if found
      Exit Function
    End If
  Next
End Function
```

The **GetPortByName** utility function – gets all Ports and matches by requested Port Name and returns if found.

Main Functionality – Create Space around an Object

```
Private Function CreateSpace(ByVal objRange As IRange, ByVal oParent As FolderSpace, ByVal strSpaceType As String,
          ByVal oCType As SpaceConstructionType) As SpaceBase

  'Create the Space Range as 10% extra on each edge -  Arrive at diagonal ends of such expanded range
  Dim oPtLow, oPtHigh, oPtCtr As Position, oVec As Vector
  oPtLow = New Position(objRange.Range.Low)
  oPtHigh = New Position(objRange.Range.High)
  oPtCtr = New Position(0.5 * (oPtLow.X + oPtHigh.X), 0.5 * (oPtLow.Y + oPtHigh.Y), 0.5 * (oPtLow.Z + oPtHigh.Z))

  oVec = oPtCtr.Subtract(oPtLow) ' Get vector from center to Low Point
  oVec.Length = 1.1 ' elongate the vector by 10% in magnitude
  oPtLow = oPtCtr.Offset(oVec) ' arrive at the expanded range end point
  oVec = oPtCtr.Subtract(oPtHigh) ' Get vector from center to High Point
  oVec.Length = 1.1 ' elongate the vector by 10% in magnitude
  oPtHigh = oPtCtr.Offset(oVec) ' arrive at the expanded range end point

  ' Create Space by 2 points
  Dim oCatBaseHelper As New CatalogBaseHelper, oSpaceBase As SpaceBase
  Dim oSpaceInputs(1) As SpaceInputs, oInputsColl As New Collection(Of SpaceInputs)

  ' Prepare SpaceInputs for creation
  oSpaceInputs(0) = New SpaceInputs
  oSpaceInputs(1) = New SpaceInputs
  oSpaceInputs(0).InputObject = New Point3d(ClientServiceProvider.WorkingSet.ActiveConnection, oPtHigh)
  oSpaceInputs(1).InputObject = New Point3d(ClientServiceProvider.WorkingSet.ActiveConnection, oPtLow)
  oInputsColl.Add(oSpaceInputs(0))
  oInputsColl.Add(oSpaceInputs(1))

  ' Create appropriate Space object : Area / Interference / Zone
  Select Case strSpaceType
    Case "AREATYPE" : oSpaceBase = New AreaSpace(oCatBaseHelper.GetPart("SPACE_DEF_SA01"), oCType, oParent, oInputsColl)
    Case "INTERFERENCETYPE": oSpaceBase = New InterferenceSpace(oCatBaseHelper.GetPart("SPACE_DEF_IV01"), oCType, oParent, oInputsColl)
    Case "ZONETYPE" : oSpaceBase = New ZoneSpace(oCatBaseHelper.GetPart("SPACE_DEF_HZ01"),oCType, oParent, oInputsColl)
    Case Else :  Return Nothing ' Error
  End Select

      oSpaceBase.AssociatedObject = objRange ' associate the object with this space
  oSpaceBase.SetUserDefinedName(oCType.ToString & "-" & strSpaceType)
  ClientServiceProvider.TransactionMgr.Compute()
  Return oSpaceBase

End Function
```

Get Range of Object and diagonally expand it by 10% on each side, using simple Point and Vector Math API.

Prepare Input Points for Space Constructor (by two points) – Points are created in active connection (Model).

Using CatalogBaseHelper, Get Space part given Part number.

Set associated object for the space object.

Main Functionality – Space Folder Creation

Create a FolderSpace object using constructor, passing RootSystem as argument.

```
'This function creates a Space Folder with given name
Private Function CreateSpaceFolder(ByVal strName As String) As FolderSpace
  Dim oSpaceRoot As BusinessObject, oSpaceFolder As FolderSpace
  ' Get Space Root of ActivePlant
  oSpaceRoot = (MiddleServiceProvider.SiteMgr.ActiveSite.ActivePlant.PlantModel.RootSystem)
  ' Create a Space Folder with given name
  oSpaceFolder = New FolderSpace(oSpaceRoot)
  oSpaceFolder.SetUserDefinedName(strName)
  Return oSpaceFolder
End Function
```

To keep it at a medium level of overview, this sample did not include all other application areas already implemented so far – For example, it didnt cover Structure and Grids, which also have API .

The command sample is now complete – we can run it.

S3D .net API

Application Functionality

–

Grids and Structure

Grids

Assembly: GridsMiddle.dll

Classes: GridAxis, GridElevationPlane, GridPlane, GridLine, GridArc, GridCylinder

(**Note**: CoordinateSystem is defined in CommonMiddle.dll)

Entire coordinate system construction workflow:

1. Create the Coordinate System:

```
Dim oCS As CoordinateSystem
oCS = New CoordinateSystem(oModelConn, CoordinateSystem.CoordinateSystemType.Grids)
```

2. Create the grid axis X,Y,Z

```
Dim oXAxis As New GridAxis(oCS, AxisType.X)
Dim oYAxis As New GridAxis(oCS, AxisType.Y)
Dim oZAxis As New GridAxis(oCS, AxisType.Z)
```

3. Create Elevation Planes – inputs are the GridAxis perpendicular to the plane (Z) and value (elevation):

```
Dim oElevPlane_Z_0 As New GridElevationPlane(0.0, oZAxis)
```

Grids

4. Create Grid Planes:

```
Dim oGridPlane_X_0 As New GridPlane(0.0, oXAxis)
Dim oGridPlane_Y_0 As New GridPlane(0.0, oYAxis)
```

5. Create Grid Lines as the intersections between Elevation Planes and Grid Planes:

```
Dim oGridLine_Z0_X0 As GridLine = oElevPlane_Z_0.CreateGridLine(oGridPlane_X_0)
Dim oGridLine_Z0_Y0 As GridLine = oElevPlane_Z_0.CreateGridLine(oGridPlane_Y_0)
```


Structure

Assembly: SmartPlantStructureMiddle.dll

Classes: MemberSystem, MemberPart, Slab, WallSystem, WallPart, Stair, Ladder, Handrail, EquipmentFoundation, Footing, FrameConnection

- **Available functionality:**
  - Ability to place Members, Traffic Entities (Stairs, Ladders, Handrails), Foundation Entities (Footings, Equipment Foundations), Walls using Sketch3D, ,Slabs bounded by Sketch3D,
  - establish connectivity with these objects using topology ports.
- **What's not yet available in v2009.1:**
  - Ability to place detailing objects (Openings, Features, Assembly Connections, Secondary Parts), Walls and Slab boundaries by Sketch2D, Analysis objects.

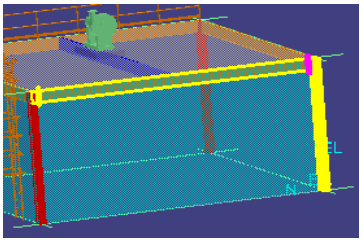
Structure Entities

- **Profile Group –** Members
- **Connection Group –** Frame Connection , Split Connection
  - Assembly & Physical Connections → Not in 2009.1
- **Area Entities Group –** Slab
- **Wall Group –** Walls
- **Port Group –** MemberSystemAxisPort , MemberPartAxisPort , TopologyPort
- **Traffic Entities Group –** Ladder, Stair, Handrail
- **Foundation Entities –** Footings, Equipment Foundations

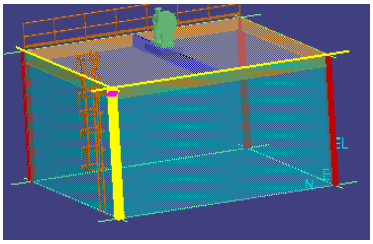
Profiles - MemberSystem

- **Logical collection of member parts. Maintain the design basis and physical alignment of the member parts for analysis, design and manufacturing.**
  - On creation, also creates a single MemberPart and two Frame Connections, one at either end.
  - Geometry represents the 'axis along' port.
- **Common arguments to all constructors (except Copy Constructor)**
  - ParentSystem, CrossSection, Material, MemberType, CardinalPoint, RotationAngle, MirrorFlag
- **Different Constructors based on the construction technique**
  - **Starting and Ending Positions** → Linear member, Unconnected at ends
  - **Sketch3D** → Curved Member, Unconnected at ends
  - **Copy Constructor** → Exact copy of the source member (connected to same related objects)
  - **Starting and Ending RelatedObject Collections** → One or more constraining objects in a collection should provide a unique position in space:

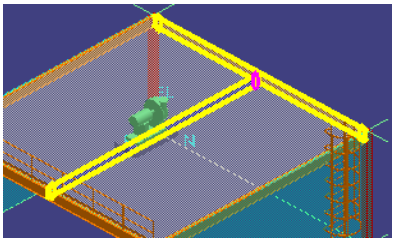
Another Member's Frame Connection



Intersecting GridLines



A MemberSystem and a Point3D on it



MemberSystem, FrameConnection

MemberSystem

Important Methods/Properties

- Get/Set RelatedObjects at a given end
- Get Parts
- Get/Set End Points
- Get FrameConnection / SplitConnection
- ICurve interface (Intersect, DistanceBetween) – for figuring out topology

Frame Connection

Describes positioning relationship between the parent MemberSystem and constraining objects

- Generated by the parent MemberSystem
- Symbol based, so user customizable
- Supports part selection logic

- **Important Methods/Properties**
  - Get/Set RelatedObjects
  - Get/Set Part
  - ValidateRelatedObjects
  - ExecuteSelectionRule e.g. oFCStartSupported.ExecuteSelectionRule("Seated-Top")

Area Entities - Slab

- **Used to model solid surfaces in the model**
  - Consists of one or multiple layers as defined in the SlabComposition
- **Creation → Using Constructors -** Inputs include ParentSystem, PlaneDefinition, Sketching Plane, BoundaryDefinition Collection, SlabType Part, SlabComposition Part, Thickness
- **Copy Constructor →** Exact copy of the source slab (bounded by same objects)

PlaneDefinition – holds Data defining the slab base surface. Provides ability to create planes in various ways. The various Creation Methods Supported are

1) Coincident with another plane or surface; **2)** Offset from another plane or surface

3) Specified by 3 points. **4)** Specified by Point and normal vector (defined by two points)

5) Rotated at an angle to another plane or surface; rotation axis (specified by start/end points) has to be parallel to the referenced plane.

BoundaryDefinition – used to define inputs for each boundary. Also specifies specific faces of those inputs and offsets. Can specify using the below possible inputs:

- Member parts, GridPlanes, GridLines

- **Important Methods/Properties**
  - Resolve Ambiguity
  - Get Layers (see information on Layer below)
  - FacePosition (Whether the PlaneDefinition is Bottom, Center, Top)
- **Layer →** Provides material and thickness information for a single layer in the SlabComposition.

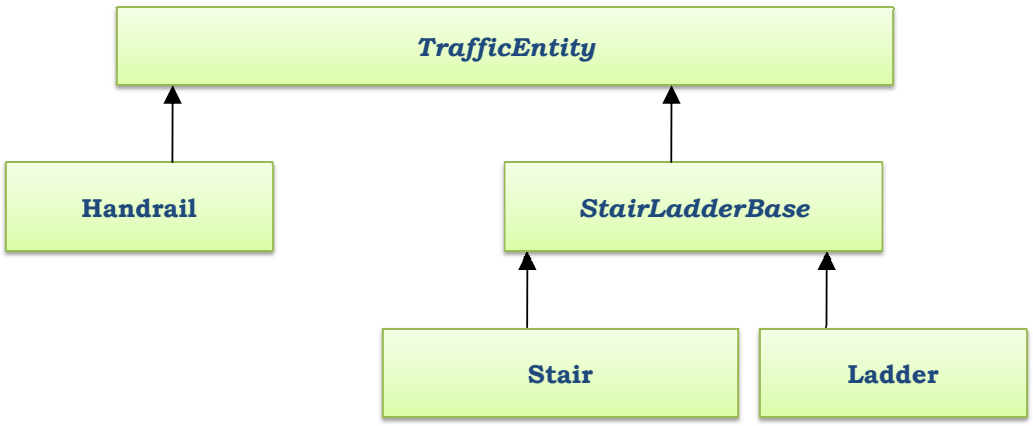
Wall Entities

- **Wall Abstract Class**
  - Provide generalization of all entities supporting plate like behavior and functionality
  - Implements INameRule, ISystem, ISystemChild, IRange, ITransform, IWBSItemChild, IEDIData, IConnectable and IStructConnectable.
- **Wall System**
  - Logical container for wall part and run.
  - Surface representation of a wall, obtained by extruding the path
  - On creation, also creates a wall part and run.
- **Creation → using Constructors**
  - Different Constructors based on the construction technique → By Sketch3D Path
  - **Copy Constructor →** Exact copy of the source Wall System
  - **Common arguments to all other constructors (except Copy Constructor)** ParentSystem, CrossSection, Wall Catalog Part, Wall Composition Part, Support Plane, Cardinal Point, Reflect flag, Height, Thickness
- **Important Methods/Properties →** Get/Set Boundaries
- **Wall Part**
  - Represents the real physical part in the model
  - Geometry is represented by a solid
  - Created by the parent Wall System.

Ports

- **Port**
  - Represents a proxy for a piece of geometry from the composite geometry of an entity
  - Hidden objects which can be related to
  - PortTypes – Face, Edge, Curve
- **StructPortBaseAbstract Class**
  - Provide generalization of all entities supporting port like behavior and functionality
  - Realizes Port behavior
- **MemberPartAxisPort**
  - Ports on MemberPart axis geometry
  - Start (Vertex), End (Vertex), Along (Edge)
- **MemberSystemAxisPort**
  - Ports on MemberSystem axis geometry
  - Start (Vertex), End (Vertex)
  - Note: MemberSystem itself is the along port
- **TopologyPort**
  - Ports on the actual geometry of the entity
  - Realizes Range, Surface, Curve, Plane, Line or any other geometric behavior
  - Need to know the port geometry type to see which geometric interface to use
  - API users can access these ports through methods on IStructConnectable interface.

Traffic Entities - Class Hierarchy



Implemented Interfaces : INamedItem, INameRule, ISystem, ISystemChild, IAssembly, IAssemblyChild, ITransform, IRange, IWBSitemChild, IPartOccurrence, IEDIData.

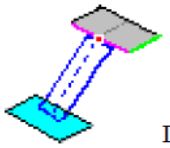
Traffic Entities – Ladder / Stair

- **Common arguments to all constructors (except Copy Constructor)**
  - ParentSystem, Ladder/Stair Catalog Part, Top Support, Bottom Support (Support: Plane, Member System)
  - Ladder/ Stair is attached to the top and bottom supports. Position 'Along' and 'WhichSide' of the top support is controlled by 'IJSPSCommonStairLadderProps::TopSupportSide' and 'ISPSHorizVertOffset ::Horizontal Offset' respectively. Those are accessed through generic property access.

    oLadder.SetPropertyValue(1.5, "ISPSHorizVertOffset", "HorizontalOffset")

    oStair.SetPropertyValue(True, "IJSPSCommonStairLadderProps", "TopSupportSide")

- **Different Constructors based on the construction technique**
  - **Without Side Reference** → Horizontal offset is from the start of the top support
  - **With Side Reference (Member System, GridLine, GridPlane, etc)** → Horizontal offset is from the intersection of the top support and side reference
  - **Copy Constructor** → Exact copy of the source Ladder/Stair

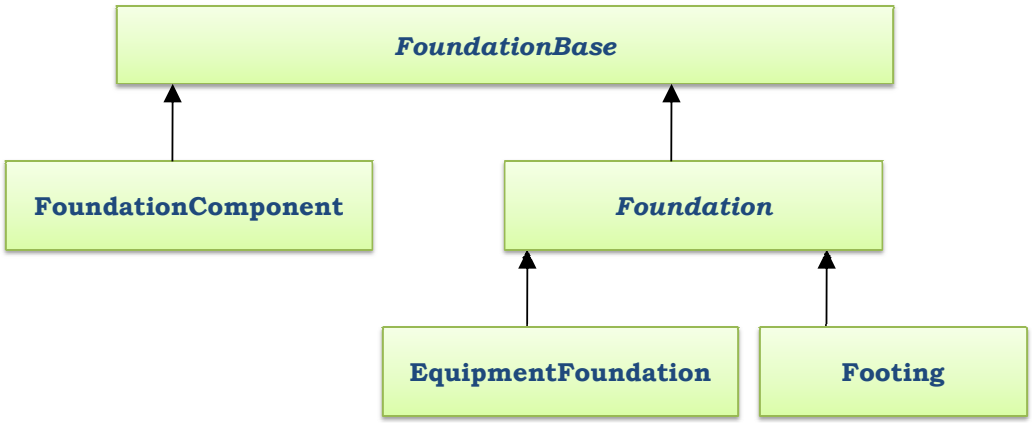


In this figure, the purple line is the top edge, the light blue plane is the bottom plane, the green line is the reference edge, and the red dot is the location point.

Traffic Entities – Handrail

- **Common arguments to all constructors (except Copy Constructor)**
  - ParentSystem, Handrail Catalog Part
- **Different Constructors based on the construction technique**
  - **By Sketch3D Path**
    - Handrail is constructed along the given path. Behavior can be customized through user attributes using generic property access.
    - Note. The Sketch path points created though the .net API are not yet Associative points – i.e. if those are selected to coincide with a Member end points, moving the member will not move the handrail
  - **Copy Constructor** → Exact copy of the source Handrail

Foundation Entities – Class Hierarchy



Implemented Interfaces : INamedItem, INameRule, ISystem, ISystemChild, IAssembly, IAssemblyChild, IRange, IWBSitemChild, IPartSelection.

Equipment Foundation and Footing also implement ITransform

Foundation Entities – Footing

- **Used for supporting a member**
- **Creation → using Constructors – Common arguments to all constructors (except Copy Constructor)**
  - ParentSystem, Footing Catalog Part
  - Cardinality (Single/Combined) of the footing is defined on the catalog part
  - Behavior in the model is customized through user attributes using generic property access
- **Different Constructors based on the construction technique**
  - **By Supported MemberSystem →** A single Footing is created below the bottom end of the member system
  - **By Supported MemberSystems →** A single or combined Footing (determined by the catalog part) is created below the bottom end of the bottom-most member system
  - **By Supported MemberSystem and Supporting Plane →** A single Footing is created below the bottom end of the member system and is bounded by the given supporting plane
  - **By Supported MemberSystems and Supporting Object →** A single or combined Footing (determined by the catalog part) is created below the bottom end of the bottom-most member system bounded by the supporting plane
  - **By Point →** A single Footing is created below the point
  - **By Points →** A single or combined Footing (determined by the catalog part) is created below bottom-most point
  - **By Supported Point and Supporting Plane →** A single Footing is created below point and is bounded by the given supporting plane
  - **By Supported Points and Supporting Object →** A single or combined Footing (determined by the catalog part) is created below bottom-most point bounded by the supporting plane
  - **Copy Constructor →** Exact copy of the source Footing

Foundation Entities - Equipment Foundation

- **Used for supporting an Equipment**
- **Common arguments to all constructors (except Copy Constructor)**
  - ParentSystem
  - Equipment Foundation CatalogPart is not required if it is defined on the supported Equipment
  - Foundation Port must be defined on the supported Equipment
  - Cardinality (Single/Combined) of the Equipment Foundation is defined on the catalog part
  - Behavior in the model is customized through user attributes using generic property access
- **Different Constructors based on the construction technique**
  - **By Supported Equipment →** A single Equipment Foundation is created below the foundation port of equipment
  - **By Supported Equipment Foundation Port →** A single Equipment Foundation is created below the foundation port
  - **By Supported Equipment and Supporting Plane →** A single Footing is created below the foundation port of equipment and is bounded by the given supporting plane
  - **By Supported Equipment Foundation Port and Supporting Object →** A single Footing is created the foundation port and is bounded by the given supporting plane
  - **By Supported Equipment and Equipment Foundation Catalog Part →** A single Equipment Foundation of the given type is created below the foundation port of equipment
  - **By Supported Equipment Foundation Port and Equipment Foundation Catalog Part →** A single Equipment Foundation of the given type is created below the foundation port
  - **By Supported Equipment, Equipment Foundation Catalog Part and Supporting Plane →** A single Equipment Foundation of the given type is created below the foundation port of equipment and is bounded by the given supporting plane
  - **By Supported Equipment Foundation Port, Equipment Foundation Catalog Part and Supporting Object →**A single Equipment Foundation of the given type is created below the foundation port and is bounded by the given supporting plane
  - **By Supported Objects and Equipment Foundation Catalog Part →** A single or combined Equipment Foundation is created below the foundation port of equipment
  - **By Supported Objects, Equipment Foundation Catalog Part and Supporting Objects →**A single or combined Equipment Foundation is created below the foundation ports of all equipment and bounded by the supporting objects
  - **Copy Constructor →** Exact copy of the source Equipment Foundation

S3D .net API

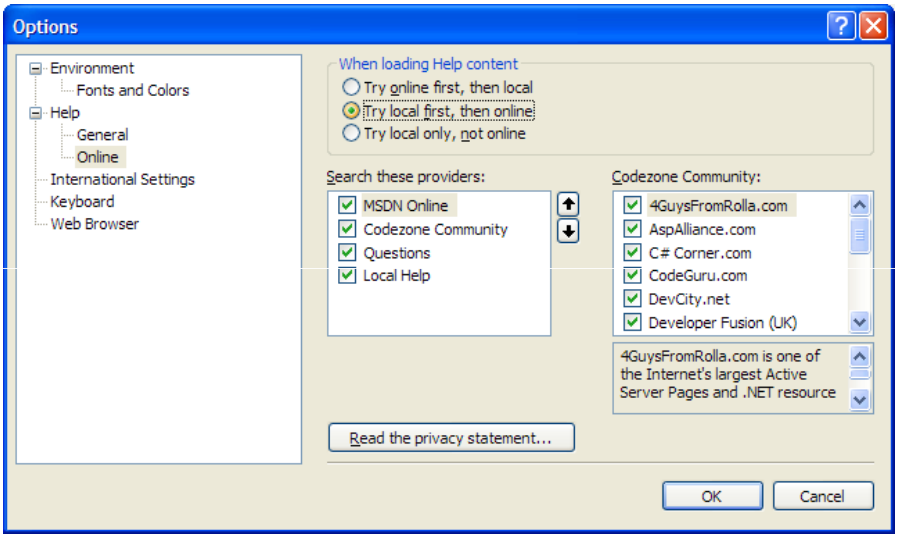
Troubleshooting

Troubleshooting

- **F1 help is not being displayed for SP3D .net classes from Visual Studio**

1. Make sure you have installed "Programming Resources". If you haven't done so, just installing it will usually fix the problem.
2. Recommend Installing it into $SmartPlant\3D\Prog\
3. Programming Resources will create "SOM" folders under each Application. Go for example under $SmartPlant\3D\Prog\CommonApp\SOM\Client\CommonClient\Programming\Help2. You will see there unregister_ and *register_CommonClient.bat* – Run (doubleclick) the unregister and then register file. You may need to do this for the other applications too.
4. Make sure your searching order for help is local and then on-line (from the search windows menu Tools/Options…)

Troubleshooting



Troubleshooting

- **I'm using Express Edition of VB.net or C# 2008 and I cannot set TaskHost.exe as my startup project for debugging.**

  If you are using the Express Edition of Visual Studio, when you go to your project Properties/Debug you will not see a "Start Action" section where you can specify "Start external program". To get around this you will need to open the file <MyProject>.vbproj.user in a text editor and add a <StartAction> and <StartProgram> nodes as shown:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <StartAction>Program</StartAction>
    <StartProgram>C:\SP3D\WS\Core\Container\Bin\TaskHost.exe</StartProgram>
    <StartArguments>C:\Documents and
Settings\Administrator\Desktop\v2009.ses</StartArguments>
  </PropertyGroup>
  <PropertyGroup>
    <ProjectView>ShowAllFiles</ProjectView>
  </PropertyGroup>
</Project>
```

Deploying .NET Assemblies on a Network share

Code Access Security

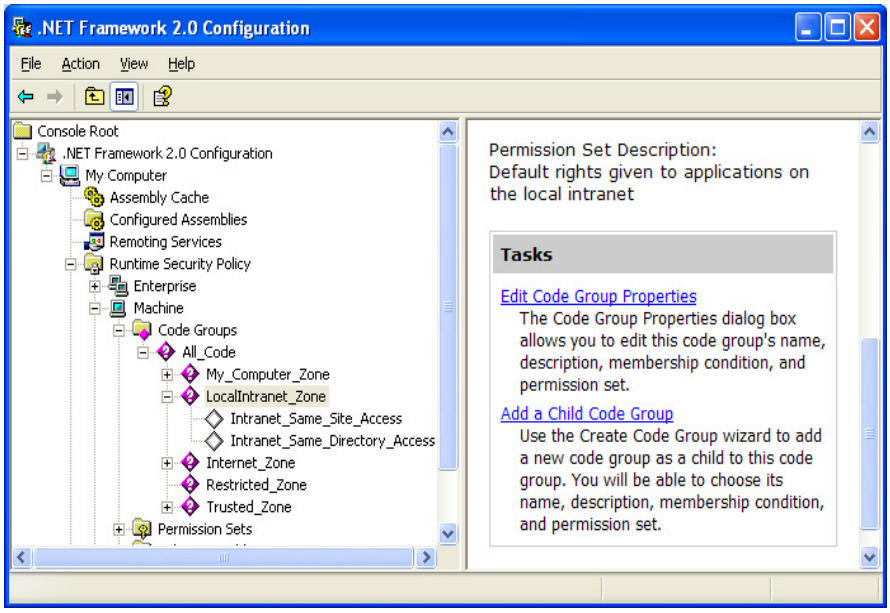
Windows security enables you to secure resources according to the role of the user. For example, an Administrator has unrestricted permissions where a Guest has very restricted access. The .NET Framework includes a security mechanism called Code Access Security that helps you to further secure your application. Using Code Access Security, different permissions are applied depending on where the code originated, identified by the Zone of the assembly and the permissions identified for that zone in the Code Access Security policy. For example, code that exists on a machine is identified as being in the MyComputer zone. Code that is downloaded from another machine on the local intranet is identified as being in the Local Intranet zone. You can apply different permission sets for each zone to provide tighter security for external code in the Local Intranet zone, while security in the MyComputer zone has fewer restrictions. By default .NET framework does not fully trust network locations and grants Local Intranet zone for assemblies loaded from a network share. If the application on network share tries to use an assembly that demands FullTrust eg: .NET Framework data providers for OLEDB, and Oracle OLEDB providers, will receive an unexpected Security Exception. The .NET Framework data providers for OLE DB, ODBC, and Oracle demand FullTrust permission when the code is run. It's easy, however, to grant this trust to your individual Assemblies, or your entire symbol folder.

You could deploy your .NET API based custom commands into a subfolder named “CustomCommands” in the SymbolShare.

On every SmartPlant 3D client machine where you want to access and use such custom commands, admin must execute the following steps to grant “FullTrust” permission to all .NET Assemblies under the symbol share folder (or any other share as you may wish) located on the network:

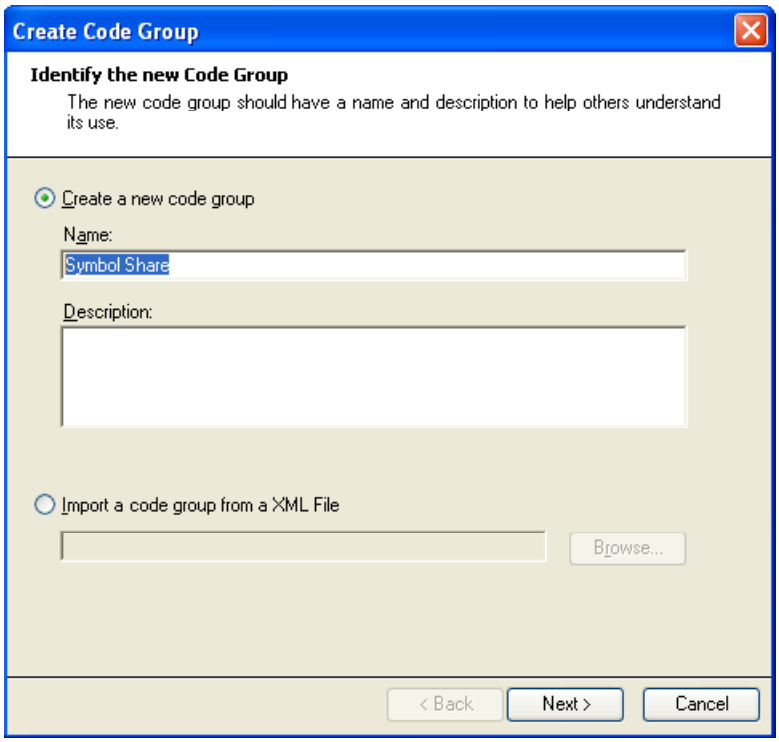
1. Open **Microsoft .NET Framework Configuration** which you'll find under **Administrative Tools** in the **Control Panel**.

2. Expand **Runtime Security Policy > Machine** > **Code Groups > All_Code > Local Intranet_Zone**



3. In the right-hand pane, click **Add a Child Code Group.**

4. In the dialog, choose **Create a new code group** and fill in the **Name** that best describes the code group and click **Next**.

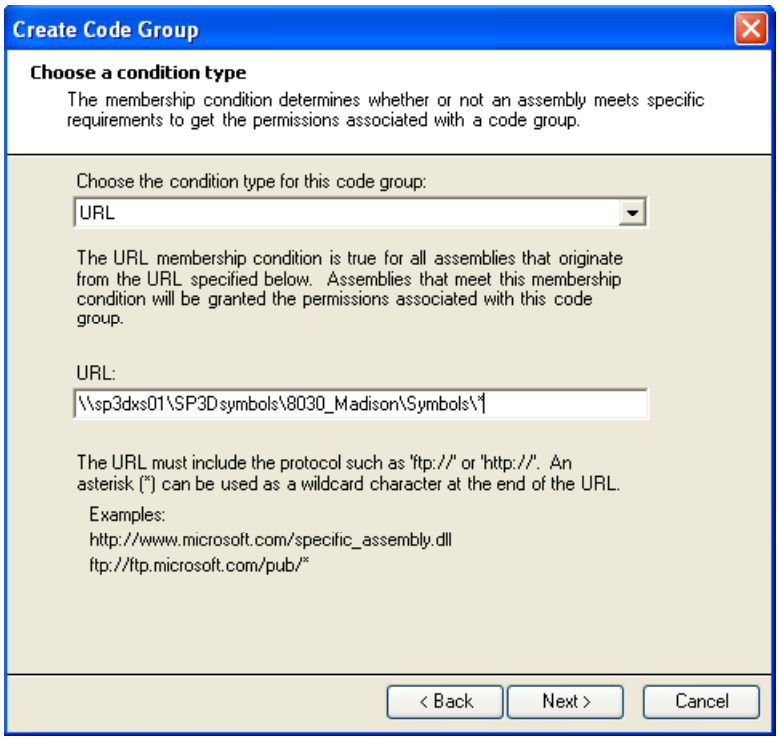


5. In the Condition Type drop down, choose URL, and

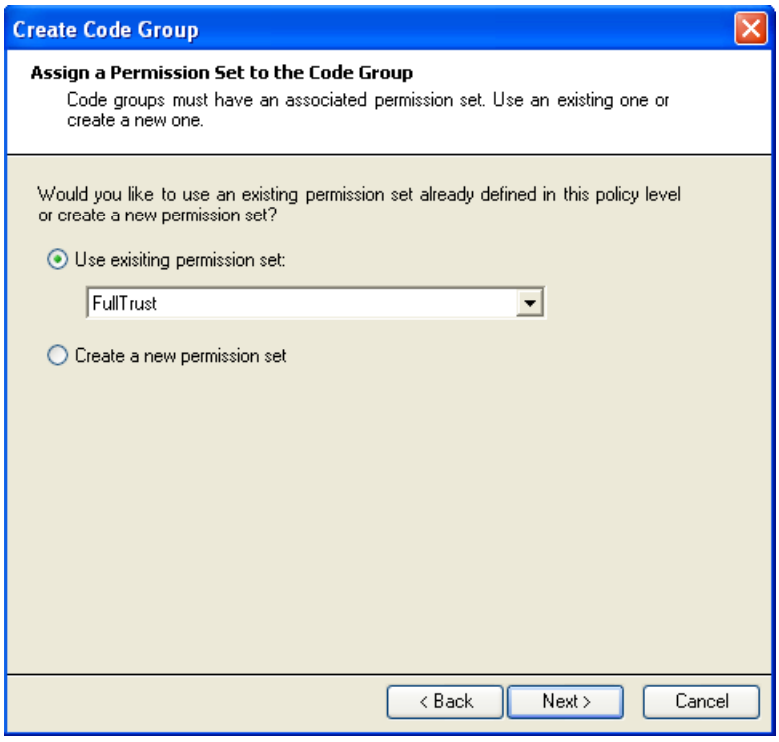
For the **URL** field, type something like this:

\\YourServer\\Your symbol share name*

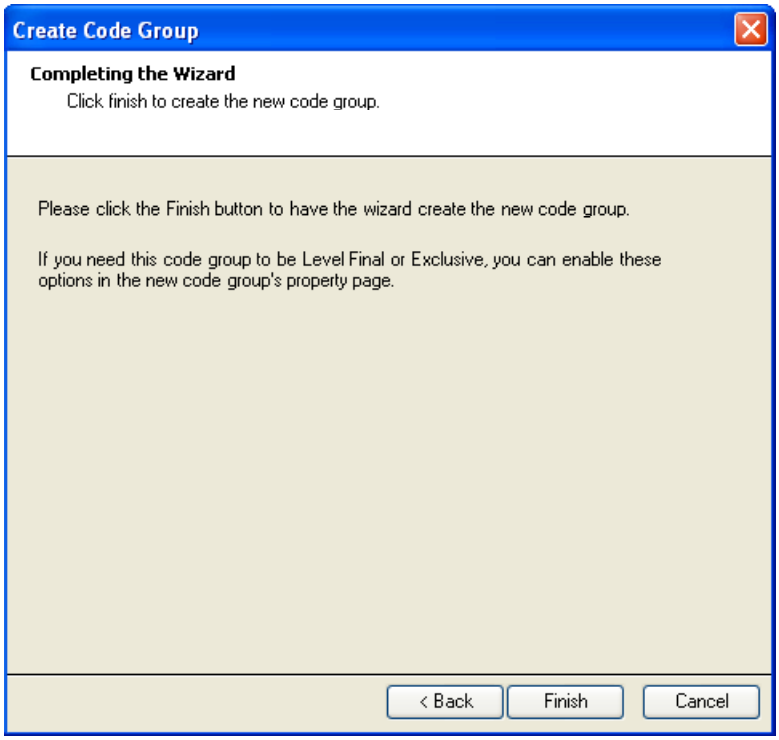
Click **Next** after you type in the share name.

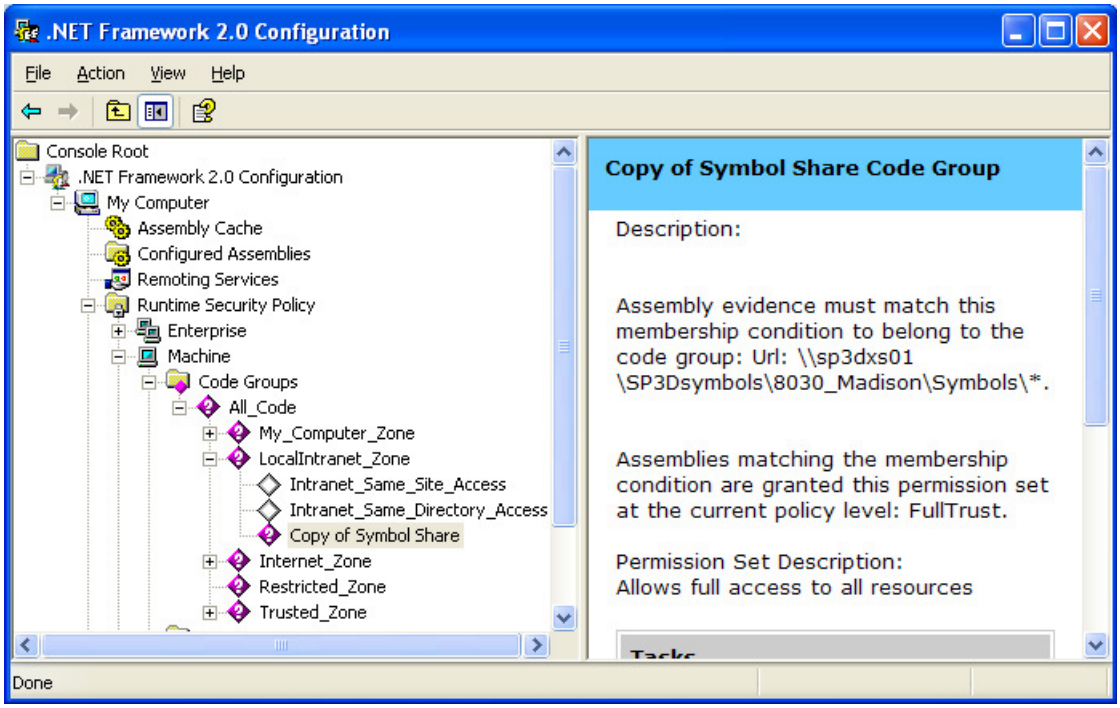


6. Under **Use existing permission set**, choose **FullTrust** and click **Next**.



7. Click Finish to complete the process of granting Full Trust to the symbols share.





Alternatively, you can perform the same operation by executing the following command in .NET command prompt window.

```
CasPol.exe -pp off -m -ag 1.2 -url file://servername/folder/* FullTrust
```