# Plugin Basics

Plug-in Basics

# What is a Plugin?

Architecture of Plugin Implementation

How to Create Detailing Components

Cases of Model-Plugin Usage

# Objective

- Understand the basics of Plug-ins
  - Plug-in types
  - How to define each type
  - How to add your Plug-in to Tekla Structures
  - How to debug a Plug-in
- Understand Plug-in logical structure
  - StructuresData
  - Input

# Definition of a plugin

- *"A small piece of software that adds a feature to a larger program or makes a program work better"*

  *- Merriam-Webster*

- Host application needs to specify an API for the plugin

- In Tekla Structures, plugins must conform to certain Tekla interfaces

# Plug-in Types

- PluginBase
  - Like a generic component (Stairs, ladders, truss, etc.)
  - Input can be freely defined
    - Any number or type of objects
    - Any number of points
  - Can be dependant or non dependant on input.
- ConnectionBase
  - Details (base plate, lifting hooks, etc.)
    - One part & one point
  - Connections (clip angle, bracing, welded joints, etc.)
    - One or more secondary parts

# Characteristics of Tekla plugins

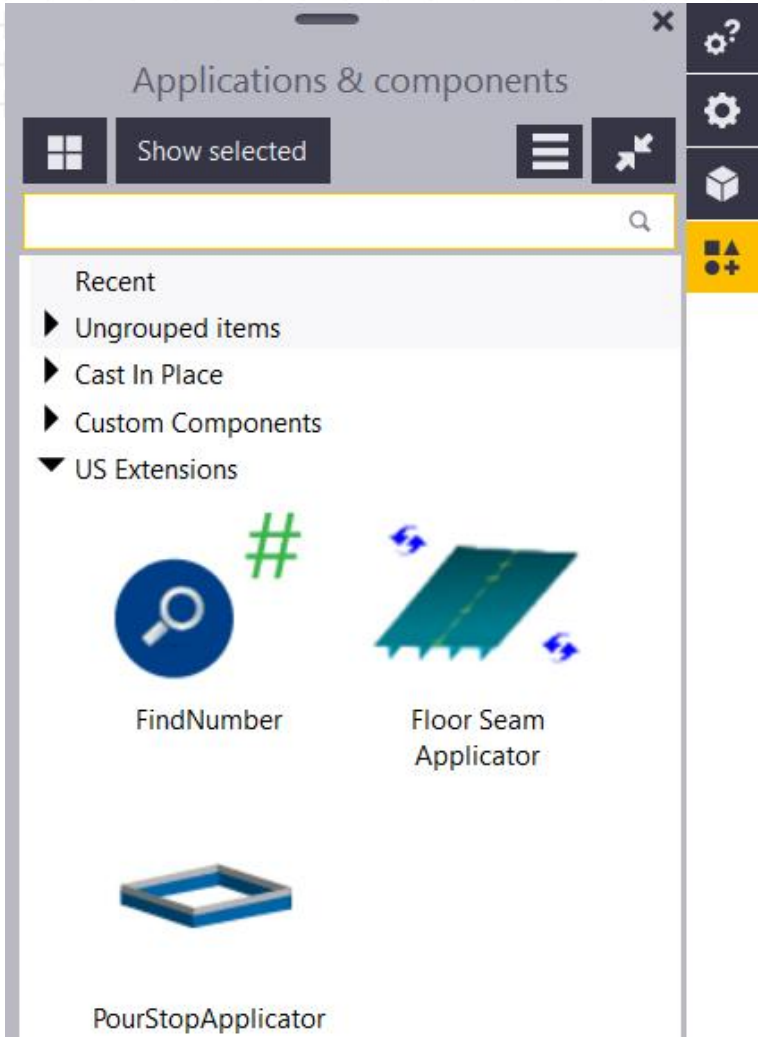
- A custom entity in the model (component)
- Has a dialog of its own
- Can create new model objects (even other plugins)
- Can be a connection, detail or generic component

# Benefits of using plugins

- Plugins are an extremely powerful way to customize Tekla Structures
- Model API is robust, reliable and comprehensive
- Tekla uses internally the same API to provide out-of-the-box connections, details and components

# Accessing plugins

- Component Catalog
- Installed
- Downloaded



# Issues to consider before starting

- Unlike Custom Components, plugins need programming skills, preferably C#
- Might be overkill for simplest needs
  - Instead record macros, use custom components
- Need good understanding of Tekla modeling system
  - Programming interfaces are very similar to Tekla model object user interfaces

What is a Plugin?

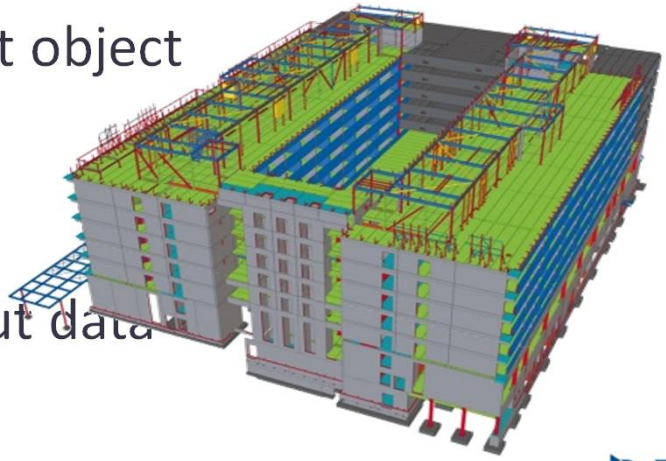
Architecture of Plugin Implementation

How to Create Detailing Components

Cases of Model-Plugin Usage

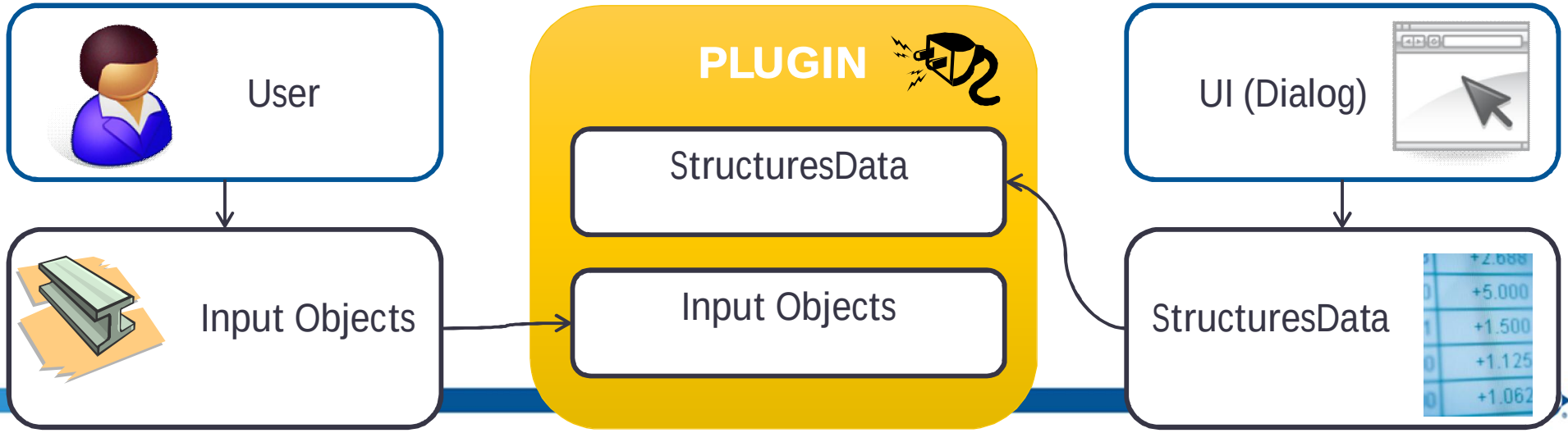
# Tekla Structures Plugin Architecture

- Inserting plugin into a model
  - Input sequence, user interface attributes
- Running a plugin
- Modifying a plugin
  - Via user interface or by dependent input object
- What is not allowed
  - Changing existing objects, changing input data



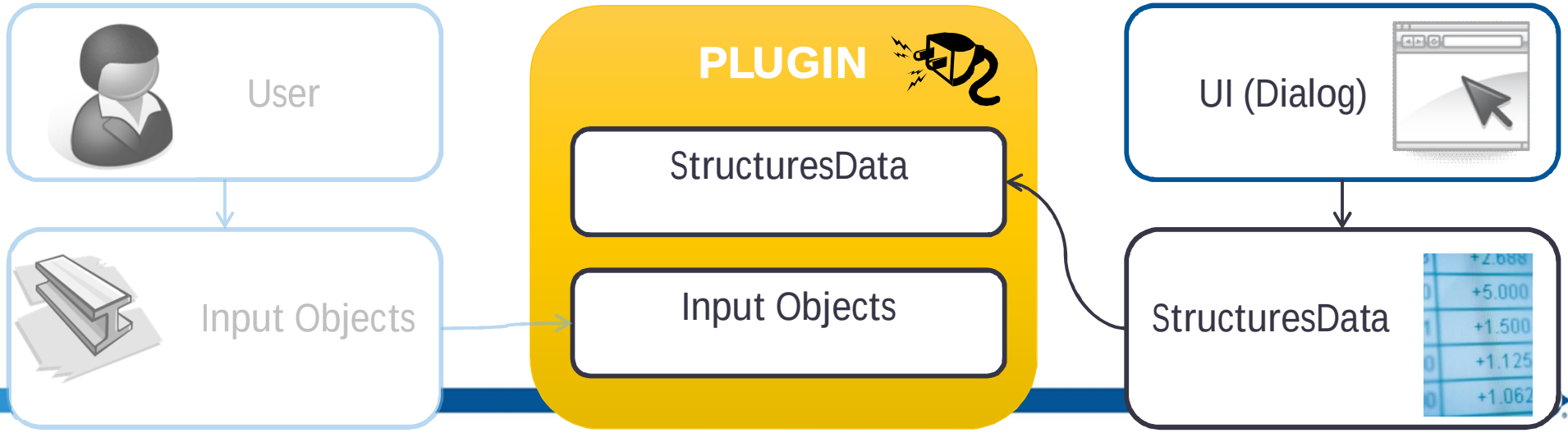
# Inserting new plugin into model

- New Plugin is started
  - Constructor method runs
  - Input prompted from user
    - Applied values taken from dialog
    - Plugin Run() when input complete
  - Both the StructuresData and the Input are stored to the Tekla model database



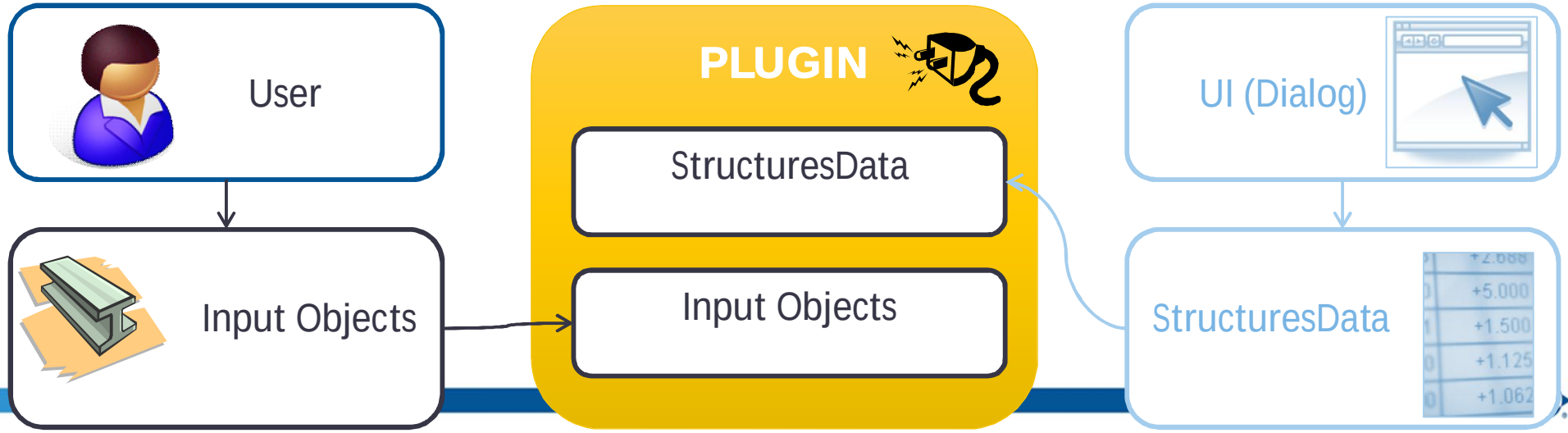
# User Interface Data Changed

- User modifies Plugin UI attributes
  - New values are read from dialog
  - Dialog data passed to Run code
  - Plugin Run() executes with new data



# Plugin automatically re-runs, input was changed

- Input object is modified, moved or otherwise changed
  - Plugin Run() executes with the changed input object(s)
  - Dialog is not shown, new parameters are not read
  - E.g. Column size changed, => Cap plate gets bigger…

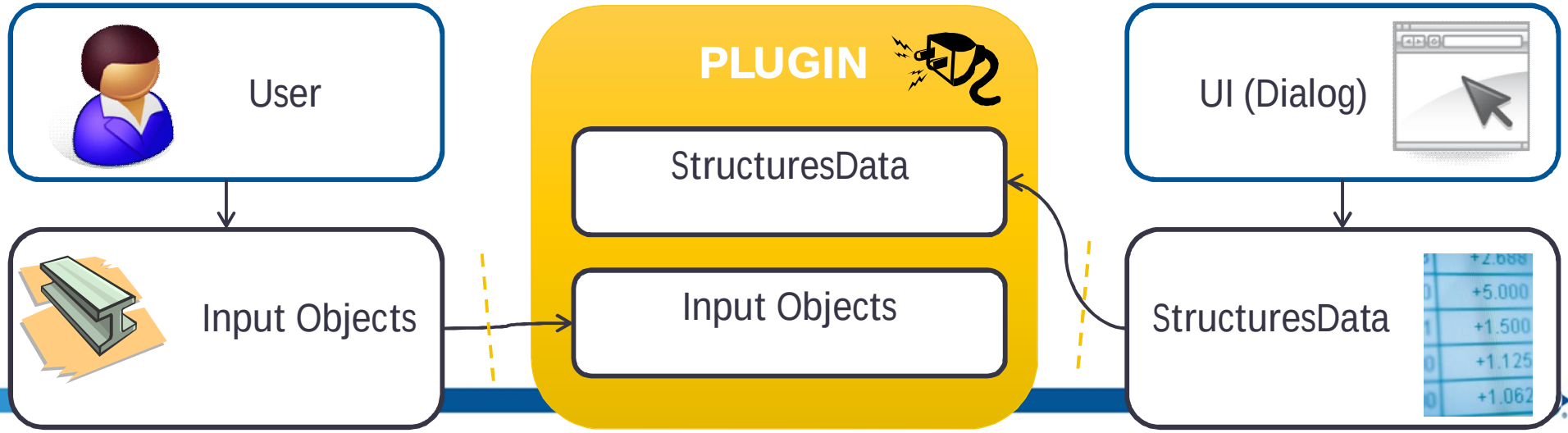


# Plugin Updates when needed automatically

- Plugin works like any system connection: If input objects are modified or user clicks ’modify’ on the dialog, it re-executes
- In general, modify works without any extra work
- Tekla system executes Run() method with modified parameters (DefineInput() is never re-executed)
- Plugin is executed just as it was added to the model
  - System automatically takes care of modifying existing parts instead of creating new ones

# Input and Data are one-way

- Plugin Run() cannot communicate via the UI
- Plugin cannot change the content of the StructuresData stored in the model (e.g. make default values explicit)
- Plugin is not allowed to modify input objects
  - Can lead to an infinite loop

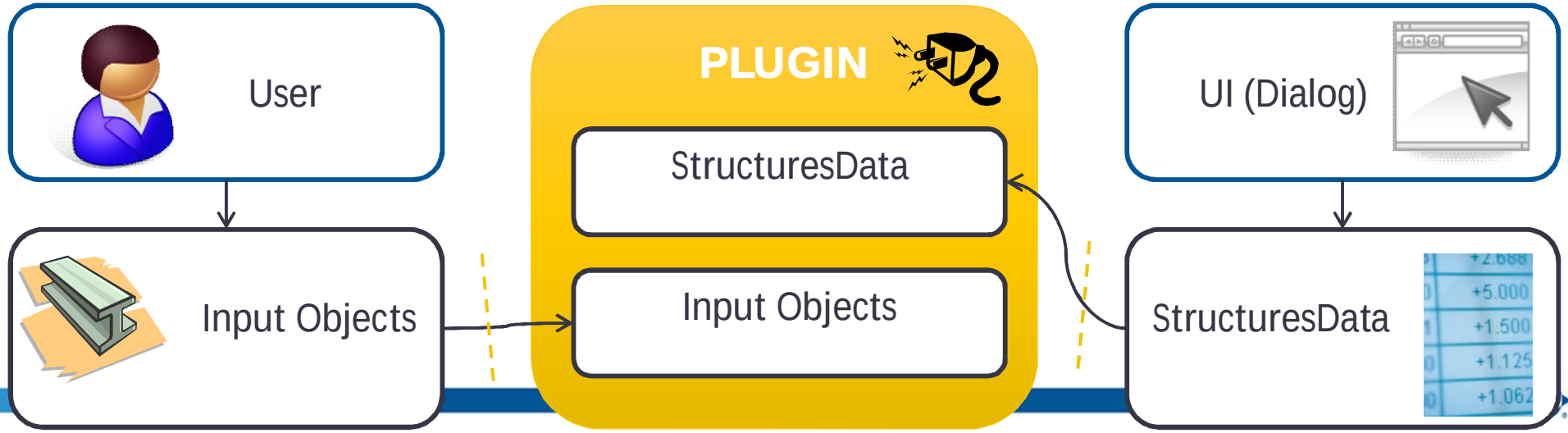


# Default values and storage

- Example
  - If attribute P1 in the dialog is left blank (default)
  - Then default hard coded value is used

```
> If(IsDefaultValue(P1))
>   P1 = 10;
```

  - The assigned default value of P1 is not stored into StructuresData



# Plugin Architecture Summary

- Two separate sources of input
  1. Input objects and points from user selection
  2. Dialog Attributes (UI)
  - Both are one-way only
  - Stored to plugin instance in model database
- DefineInput() asks user to pick points or objects to interact with from model
- Run() code
  - does work of plugin
  - cannot ask for user input

What is a Plugin?

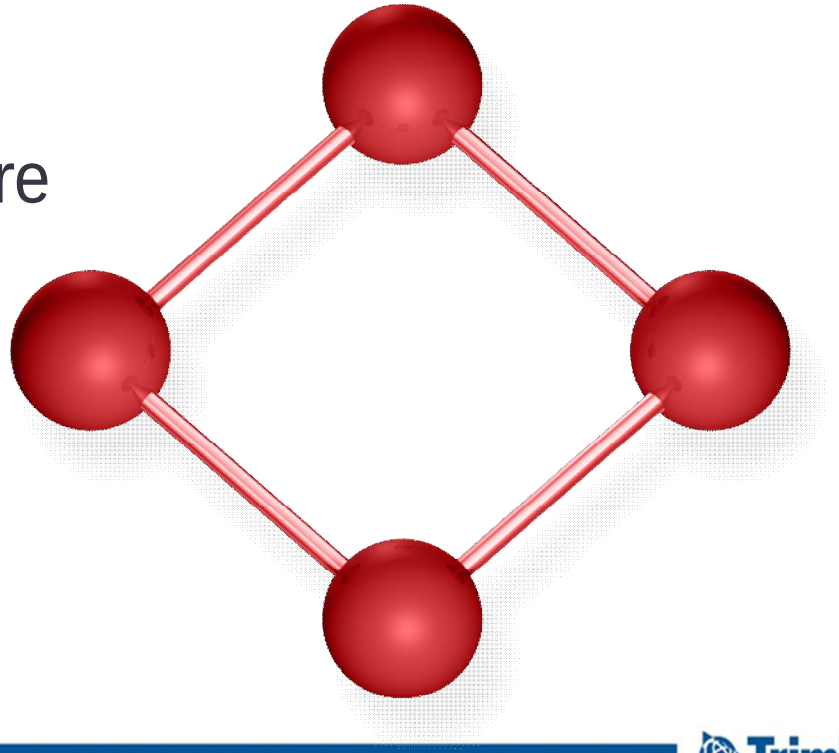
Architecture of Plugin Implementation

How to Create Detailing Components

Cases of Model-Plugin Usage

# Objectives

- Understand the basics of plugin implementation
  - Plugin types
  - How to define each type
- Understand Plugin logical structure
  - StructuresData
  - Input



# Tekla Structures Plugin Fundamentals

- All plugins need to inherit from Tekla provided base class
- Base classes are deployed in Tekla.Structures.Plugins.dll
  - Comes with Tekla Structures standard installation
- Two different types of plugins; two different ways of implementing input sequences

# Available Plugin Types

## PluginBase

- Like a generic component (Stairs, ladders, truss, etc.)
- Input can be freely defined
  - Any number or type of objects
  - Any number of points
- Can be dependent or non-dependent on input

## ConnectionBase

- Connections (clip angle, bracing, welded joints, etc.)
  - Main part and one or more secondary parts
- Details (base plate, lifting hooks, etc.)
  - Main part and one point

# Basic requirements of PluginBase

- StructuresData
  - Defines the data that can be passed from the UI
- Constructor
  - Initializes the Plug-in
  - Takes the currently applied StructuresData
- Class attributes
  - Define the name and UI
- Base class has two methods to implement
  - DefineInput()
    - Defines the input the Plug-in requires
  - Run()
    - Executes after input has been received
    - Contains plugin's "business logic"

# Plugin required attribute definitions

**[Plugin(*“PluginName1*")]**

- The name in the catalog - Must be unique

**[PluginUserInterface(*“PluginName1.MainForm”*)]**

- Points to the Form/UNP definition of the UI

```
[Plugin("FitPart")]
[PluginUserInterface("FitPart.MainForm")]
public class FitPart : PluginBase
```

```
using System;
using System.Collections.Generic;
using Tekla.Structures.Plugins;
using Tekla.Structures.Geometry3d;
using Tekla.Structures.Model.UI;
using TSM = Tekla.Structures.Model;

public class StructuresData
{
    [Tekla.Structures.Plugins.StructuresField("P1")]
    public double Parameter1;
}

[Plugin("BeamPlugin")] // Mandatory field which defines that this is the plug-in and
stores the name of the plug-in to the system.
[PluginUserInterface(BeamPlugin.UserInterfaceDefinitions.Plugin1)] // Mandatory field
which defines the user interface the plug-in uses. A Windows Forms class or a .inp file.
public class BeamPlugin : PluginBase
{
    private readonly StructuresData data;

    // The constructor argument defines the database class StructuresData and sets the
data to be used in the plug-in.
    public BeamPlugin(StructuresData data)
    {
        TSM.Model M = new TSM.Model();
        this.data = data;
    }

    //Defines the inputs to be passed to the plug-in.
    public override List<InputDefinition> DefineInput()
    {
        Picker BeamPicker = new Picker();
        List<InputDefinition> PointList = new List<InputDefinition>();

        Point Point1 = BeamPicker.PickPoint();
        Point Point2 = BeamPicker.PickPoint();

        InputDefinition Input1 = new InputDefinition(Point1);
        InputDefinition Input2 = new InputDefinition(Point2);
        PointList.Add(Input1);
        PointList.Add(Input2);

        return PointList;
    }

    //Main method of the plug-in.
    public override bool Run(List<InputDefinition> Input)
    {
        try
        {
            Point Point1 = (Point)(Input[0]).GetInput();
            Point Point2 = (Point)(Input[1]).GetInput();
            Point LengthVector = new Point(Point2.X - Point1.X, Point2.Y - Point1.Y,
Point2.Z - Point1.Z);

            if (data.Parameter1 > 0)
            {
                Point2.X = data.Parameter1 * LengthVector.X + Point1.X;
                Point2.Y = data.Parameter1 * LengthVector.Y + Point1.Y;
                Point2.Z = data.Parameter1 * LengthVector.Z + Point1.Z;
            }

            CreateBeam(Point1, Point2);
        }
        catch (Exception)
        {
        }

        return true;
    }

    static void CreateBeam(Point Point1, Point Point2)
    {
        TSM.Beam MyBeam = new TSM.Beam(Point1, Point2);

        MyBeam.Profile.ProfileString = "HEA400";
        MyBeam.Finish = "PAINT";
        MyBeam.Insert();
    }

    //.inp file user interface definition, check the Start-Up package for the Windows
Forms dialog presentation.
    public class UserInterfaceDefinitions
    {
        public const string Plugin1 = @"" +
        @"page(""TeklaStructures"",""""")" + "\n" +
         "{\n" +
         "    plugin(1, BeamPlugin)\n" +
         "    {\n" +
        @"        tab_page(""Beam test"", ""Parametri_1"", 1)" + "\n" +
         "        {\n" +
        @"            parameter(""Length factor"", ""P1"", distance, number, 1)" + "\n" +
         "        }\n" +
         "    }\n" +
         "}\n";

    }
}
```

# Basic requirements of ConnectionBase

- StructuresData
  - Defines the data that can be passed from the UI
- Constructor
  - Initializes the connection
  - Takes the currently applied StructuresData
- Class attributes
  - Define the name, UI, number of secondaries, collision type, auto up direction
- Base class has one method to implement
  - Run()
    - Executes after input has been received
    - Contains plugin's "business logic"
- Input is defined by plugin attributes, and it is managed by Tekla Structures

# Connection required attribute definitions

**[Plugin("*ConnectionName1*")]**

- The name in the catalog - Must be unique

**[PluginUserInterface(*"ConnectionName1.MainForm"*)]**

- Points to the Form/INP definition of the UI

**[SecondaryType(*ConnectionBase.SecondaryType*)]**

- The number of secondaries required
- Setting number of secondaries to zero makes it a detail

**[AutoDirectionType(*AutoDirectionTypeEnum*)]**

- Auto up direction type

**[PositionType(*PositionTypeEnum*)]**

- Connection origin type*

```
[Plugin("SpliceConnection")] //Name of the connetion in the catalog
[PluginUserInterface("SpliceConnection")]
[SecondaryType(ConnectionBase.SecondaryType.SECONDARYTYPE_ONE)]
[AutoDirectionType(AutoDirectionTypeEnum.AUTODIR_BASIC)]
[PositionType(PositionTypeEnum.COLLISION_PLANE)]
public class SpliceConnection : ConnectionBase
```

```
using System;
using System.Windows.Forms;

using Tekla.Structures;
using Tekla.Structures.Plugins;
using Tekla.Structures.Geometry3d;
using TSM = Tekla.Structures.Model;

public class StructuresData3
{
    [Tekla.Structures.Plugins.StructuresField("P1")]
    public double Parameter1;
    [Tekla.Structures.Plugins.StructuresField("P2")]
    public string Parameter2;
}

[Plugin("BeamConnection")] // The name of the connection in the catalog
[PluginUserInterface(BeamConnection.UserInterfaceDefinitions.Plugin3)]
[SecondaryType(ConnectionBase.SecondaryType.SECONDARYTYPE_ONE)]
[AutoDirectionType(AutoDirectionTypeEnum.AUTODIR_BASIC)]
[PositionType(PositionTypeEnum.COLLISION_PLANE)]
public class BeamConnection : ConnectionBase
{
    private StructuresData3 data;
    private TSM.Model M;
    public BeamConnection(StructuresData3 data)
    {
        this.data = data;
        M = new TSM.Model();
    }

    TSM.Beam CreateBeam(Point Point1, Point Point2, string Profile)
    {
        TSM.Beam MyBeam = new TSM.Beam(Point1, Point2);
        MyBeam.Profile.ProfileString = Profile;
        MyBeam.Finish = "PAINT";
        MyBeam.Position.Depth = Tekla.Structures.Model.Position.DepthEnum.MIDDLE;
        MyBeam.Position.Plane = Tekla.Structures.Model.Position.PlaneEnum.RIGHT;
        MyBeam.Insert();
        return MyBeam;
    }

    Boolean CreateFitting(Point Point1, Point Point2, double Thickness, TSM.Beam
MySecondary)
    {
        TSM.Fitting MyFitting = new TSM.Fitting();
        MyFitting.Plane.Origin = new Point(Thickness, 0, 0);
        MyFitting.Plane.AxisX = new Vector(0, 1000, 0);
        MyFitting.Plane.AxisY = new Vector(0, 0, 1000);
        MyFitting.Father = MySecondary;

        return MyFitting.Insert();
    }

    public override bool Run()
    {
        try
        {
            // The default values
            if (IsDefaultValue(data.Parameter1))
                data.Parameter1 = 300.0;
            if (IsDefaultValue(data.Parameter2))
                data.Parameter2 = "PL10*300";

            // Get secondary
            TSM.Beam Secondary = M.SelectModelObject(Secondaries[0]) as TSM.Beam;

            Point Point1 = new Point();
            Point Point2 = new Point();
            if (data.Parameter1 > 0)
            {
                Point1.Y -= data.Parameter1 / 2;
                Point2.Y += data.Parameter1 / 2;
            }

            TSM.Beam NewBeam = CreateBeam(Point1, Point2, data.Parameter2);

            double Thickness = 0.0;
            NewBeam.GetReportProperty("PROFILE.WIDTH", ref Thickness);
            CreateFitting(Point1, Point2, Thickness, Secondary);
        }
        catch (Exception e)
        {
            MessageBox.Show(e.ToString());
        }

        return true;
    }

    public class UserInterfaceDefinitions
    {
        public const string Plugin3 = @"" +
        @"page(""TeklaStructures"",""""""")" + "\n" +
         "{\n" +
         "    joint(1, BeamConnection)\n" +
         "    {\n" +
        @"        tab_page(""Beam test"", ""Parameters"", 1)" + "\n" +
         "        {\n" +
        @"            parameter(""Plate Length"", ""P1"", distance, number, 1)" + "\n" +
        @"            parameter(""Profile"", ""P2"", profile, text, 2)" + "\n" +
         "        }\n" +
         "    }\n" +
         "}\n";
    }
}
```

# Basic Requirements

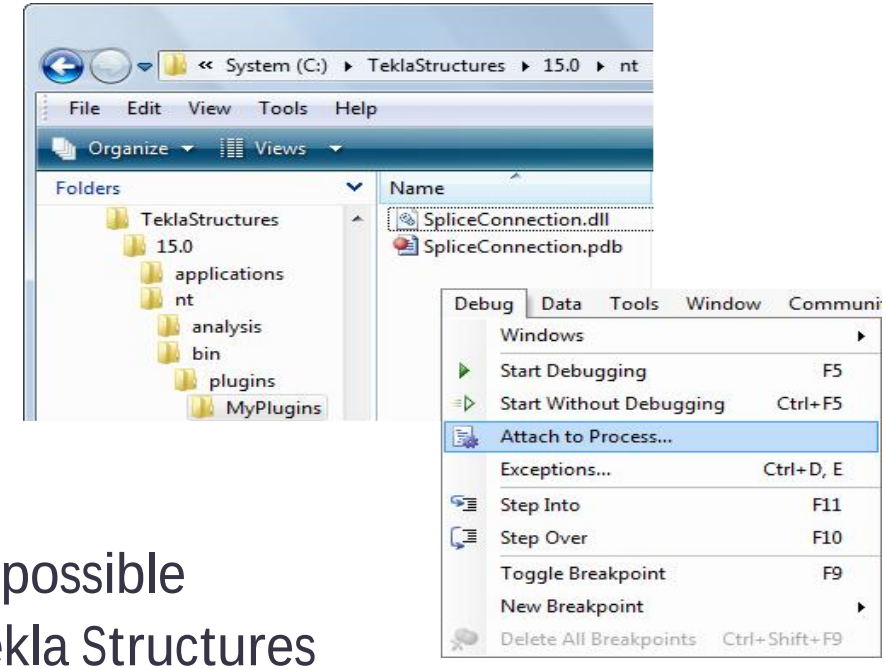
| | PluginBase | ConnectionBase |
|---|---|---|
| StructuresData | Defines the data that can be passed from the UI | |
| Constructor | Initializes & takes the currently applied StructuresData | |
| Class Attributes | Define the name & UI | Define the name, UI, number of secondaries, collision type, auto up direction |
| Run() | Executes after input has been received and contains plugin's "business logic" | |
| DefineInput() | Defines the input the Plugin requires | – |

# Plug-in dependency

- Plug-ins cannot modify their inputs.
- Plug-ins dependency can be set with the attribute InputObjectDependency.
  - Dependent: updated when input changes.
  - Non-Dependent: doesn't update when input changes.
  - Geometrically-Dependent: Plug-in updates when the input part geometry changes. This Plug-in cannot create any boolean objects to the input part, since it would cause and endless loop.
  - Non-Dependant-Modifiable*Next: No dependency on input but the instance is modifiable in the model. The created objects have a relation to the plug-in. The plug-in dialog can be opened from the created objects.

# Running and Debugging a Plug-in

- Preparation
  - Copy the project dll and pdb file to the Plug-ins folder or a sub folder
  - Run Tekla Structures
  - Set breakpoints in the code
- Debugging
  - Debug > Attach to process
  - Run or modify the Plug-in
  - Debug > Stop debugging
- Changes
  - On the fly code changes are not possible
  - A new dll requires a restart of Tekla Structures



# Plug-in User Interface

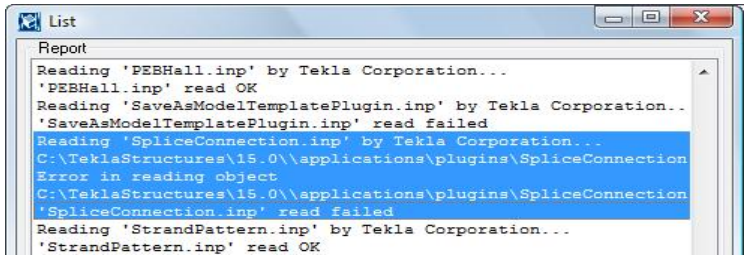
- Forms
  - Includes save & load, apply, get, etc. functionality
  - Supports selecting from catalogs (bolt, profile, etc.)
  - Support for distance (units), double, int, and string only
  - More powerful than INP.
- INP
  - Same format used for custom components & system components
  - Includes save & load, apply, get, etc. functionality
  - Supports selecting from catalogs (bolt, profile, etc.)
  - Supports all data types and type checking
  - See the online help for the details of INP files

# Notes

- Visual Studio
  - Plug-in projects are a ‘Class Library’ (e.g. dll)
- Plug-in dlls
  - More than one Plug-in can be created in the same dll under the same project
- Avoid message boxes and pop-up dialogs
  - If the input is modified, any dialogs and message boxes will be shown again for each Plug-in instance
- A Plug-in sets the workplane automatically to local coordinate system when Run is called.
  - Local coordinate system: first picked part coordinate system, or coordinate system with origin in the first picked point.

# Notes

- Do not use Model.CommitChanges() in a Plug-in
  - Executed by Tekla Structures when Run() completes
  - Would set a ‘strange’ undo point
  - Can ruin the keep ID processing
- Consistent IDs under modify
  - Tekla Structures keeps IDs the same (where possible) during modify just like system components
- Trouble shooting
  - Information about problems loading Plug-ins or problems with the dialog can be found in the session history log



# Limitations and known problems

- Component Types
  - Seams and Details can be created with ConnectionBase while CustomPartBase has currently limitation that it always requires two input points
  - Both can be done as PluginBase
- AutoDefault & AutoConnection
  - Plug-in connections cannot be used in AutoDefault or AutoConnection prior to 16.1.
  - AutoDefault works in Forms starting in 17.0

# Implementation summary

- Dialog attributes defined in code (StructuresData)
- User input is stored to model with plugin
- Two base classes available
  - PluginBase
    - Input explicitly defined in code
  - ConnectionBase
    - Fixed input like connection or detail
    - Detail or Connection Type is set in header attributes for class
- Run() code contains main logic, but can not ask for user input

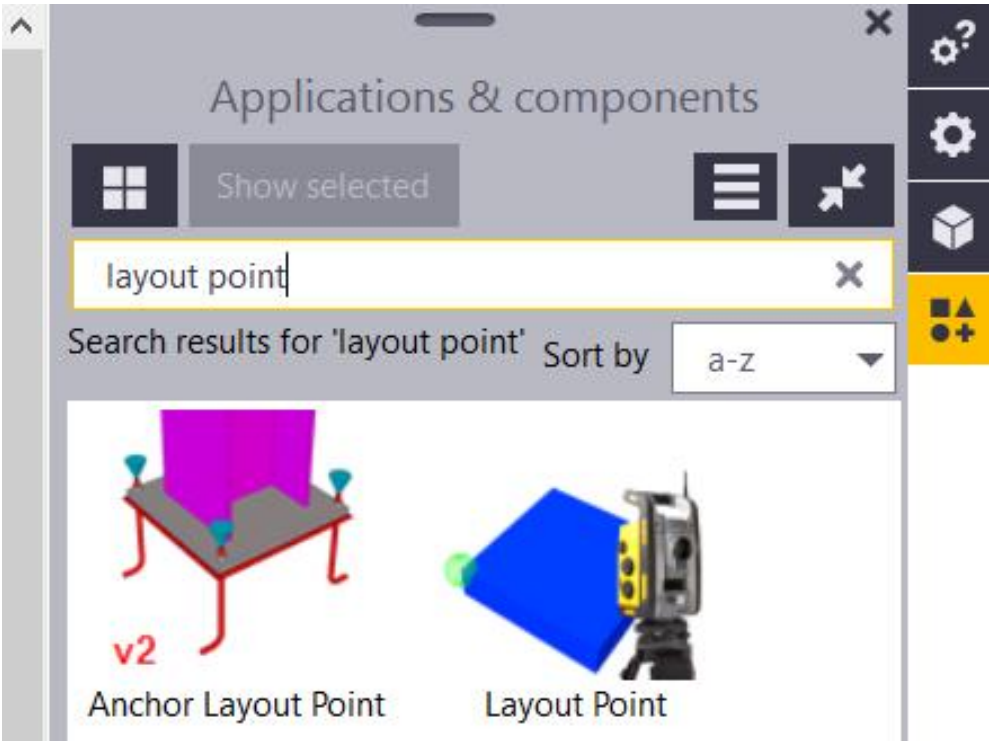
# Case: Layout Point in Depth

# Layout Point and Layout Line

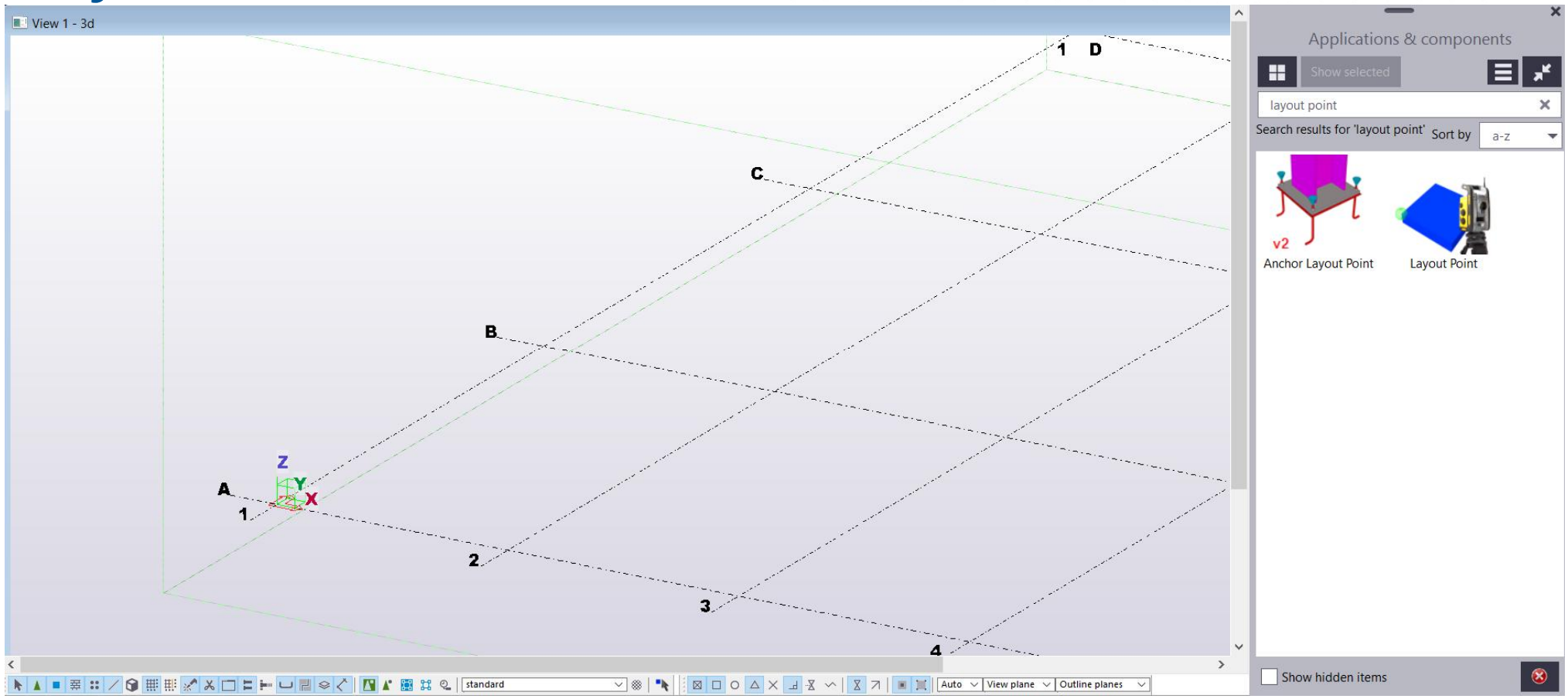
- Included in Tekla Structures installation
- Purpose is to help modeling field layout inside Tekla model
- Implemented as Plugin
- Allows user to create layout points in the Tekla model
- Layout points can in turn be used as input to Layout Lines

# Open Applications & components

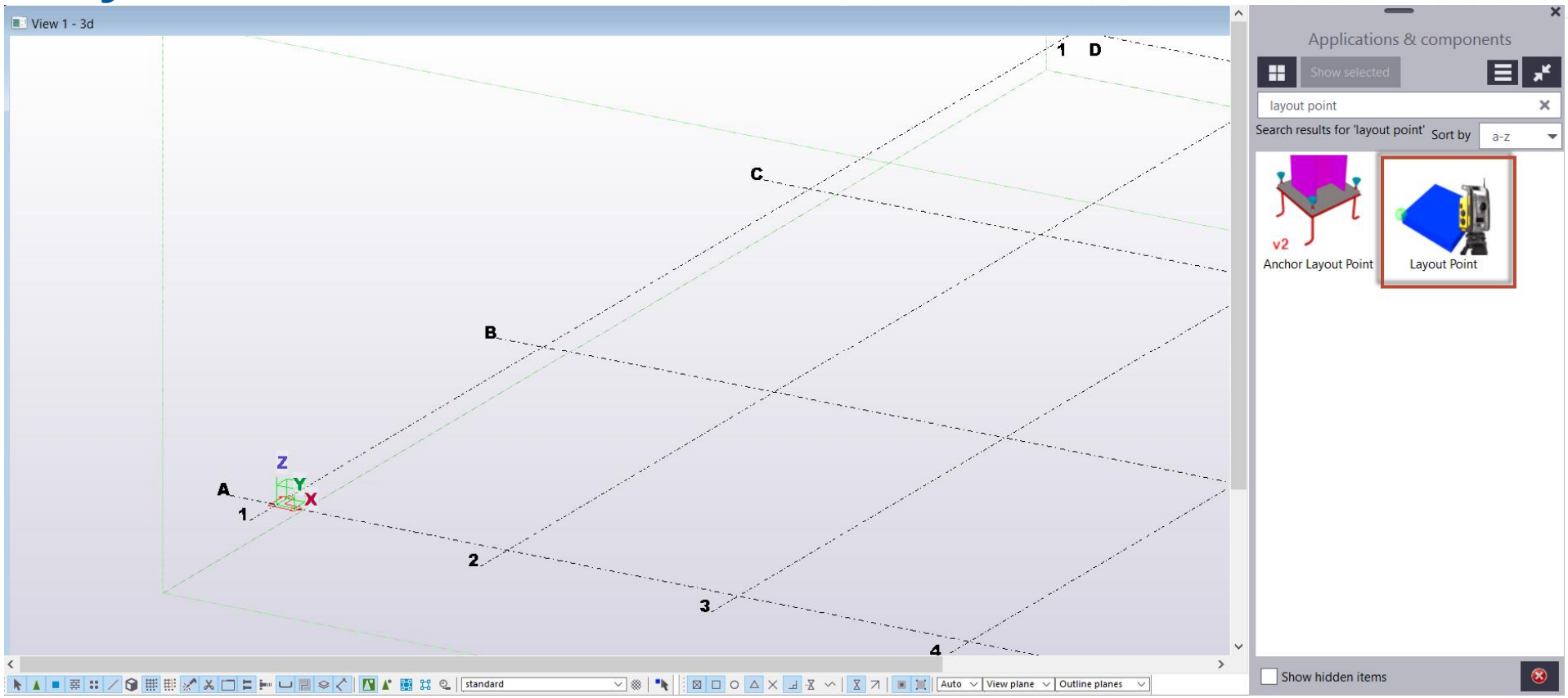
- Plugins
- LayoutPoint and LayoutLine



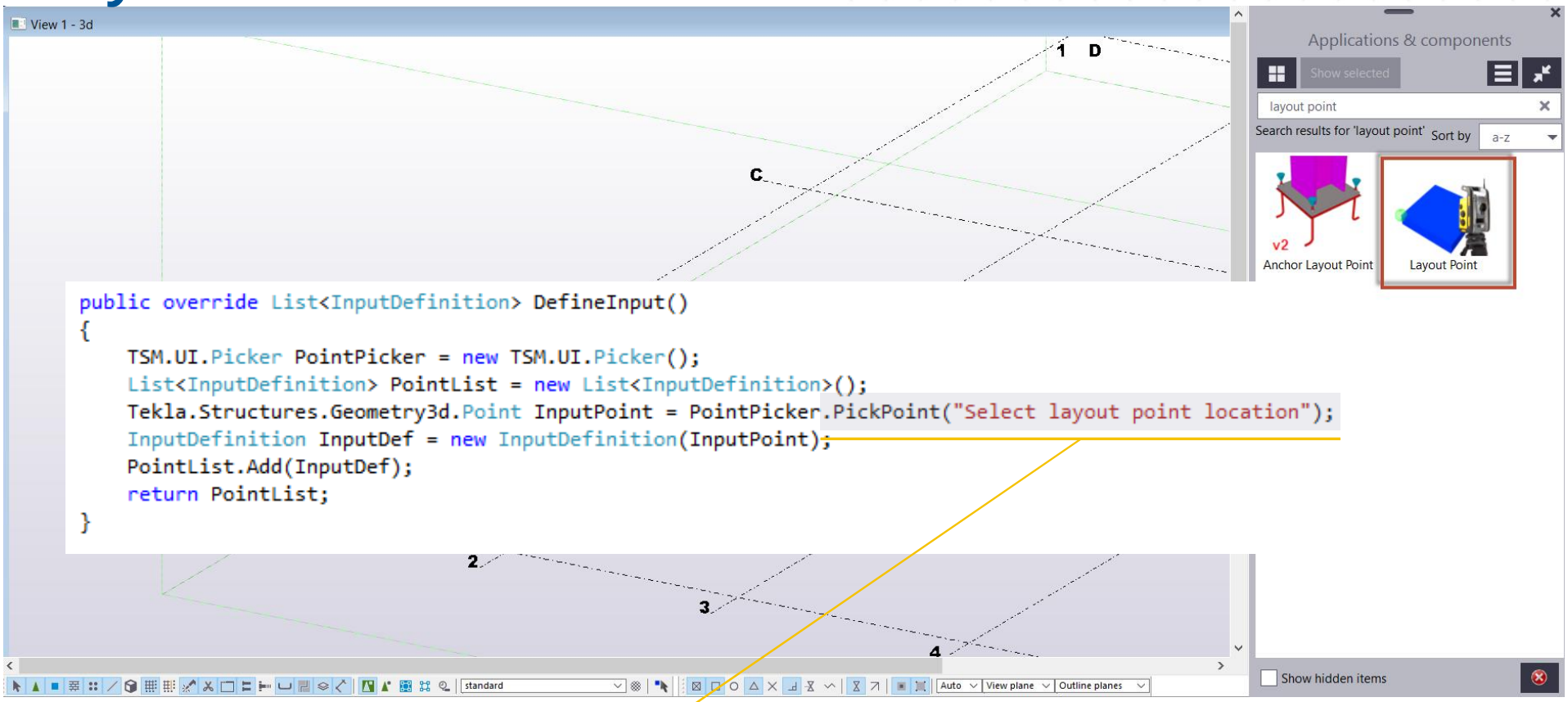
# Layout Point



# Layout Point

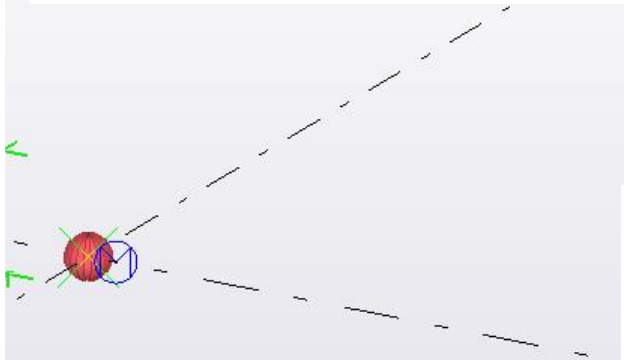


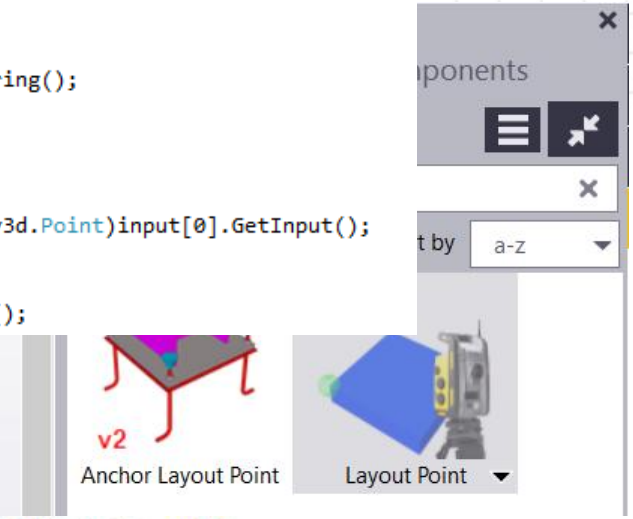
# Layout Point




```
public override List<InputDefinition> DefineInput()
{
    TSM.UI.Picker PointPicker = new TSM.UI.Picker();
    List<InputDefinition> PointList = new List<InputDefinition>();
    Tekla.Structures.Geometry3d.Point InputPoint = PointPicker.PickPoint("Select layout point location");
    InputDefinition InputDef = new InputDefinition(InputPoint);
    PointList.Add(InputDef);
    return PointList;
}
```

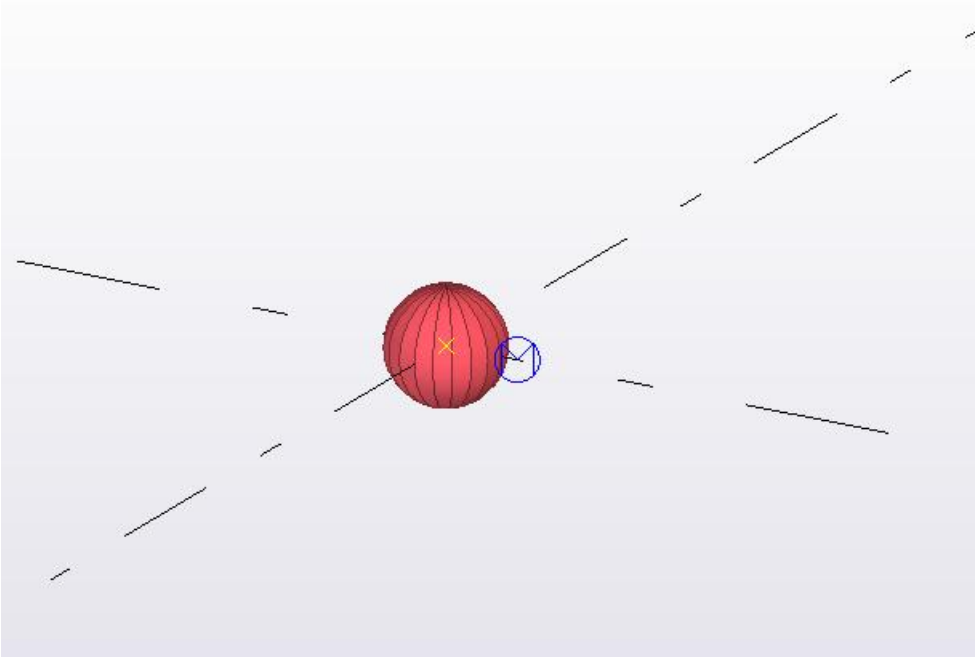
```
public override bool Run(List<InputDefinition> input)
{
    try
    {
        string UsedProfileString = "SPHERE";
        string SizeString = Tools.PLUGIN_PROPERTY_POINT_SIZE_DEFAULT_VALUE.ToString();
        ManageDefaultValues();

        Tekla.Structures.Geometry3d.Point StartPoint = null;
        Tekla.Structures.Geometry3d.Point EndPoint =null;
        Tekla.Structures.Geometry3d.Point PickPoint = (Tekla.Structures.Geometry3d.Point)input[0].GetInput();
        double size = 0.5 * _data.PointSize;
        double FullSize = _data.PointSize;
        Tekla.Structures.Model.Profile pro = new Tekla.Structures.Model.Profile();
```



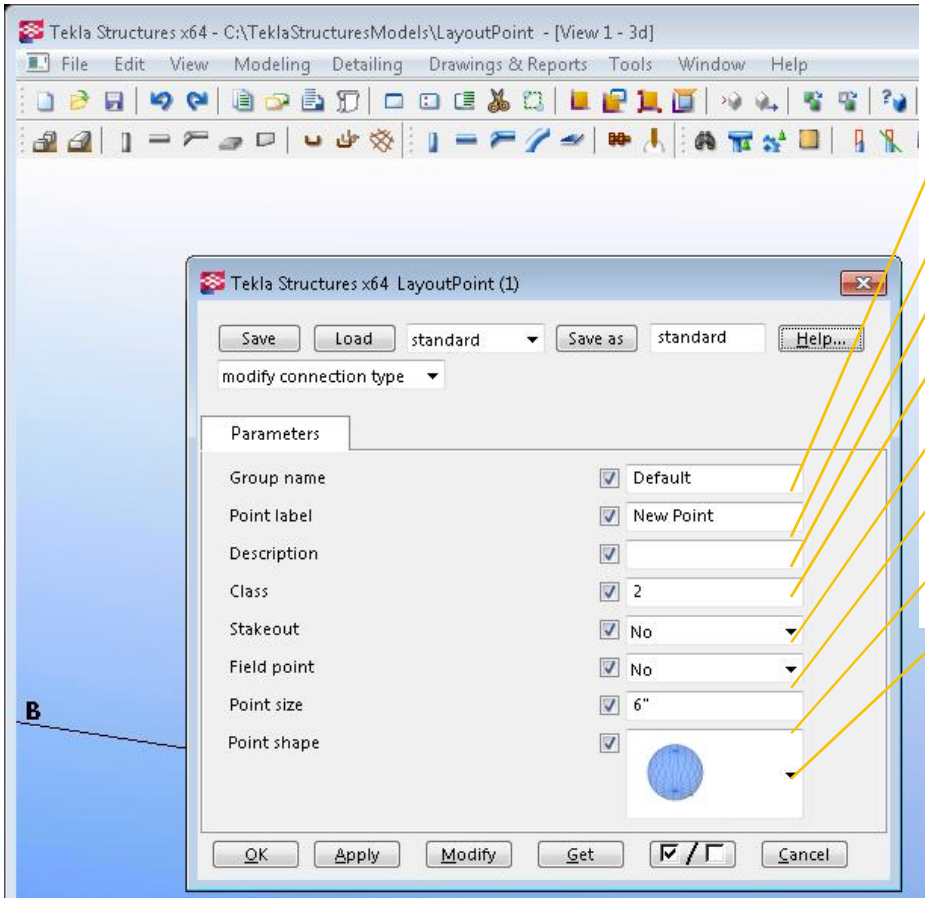



```
        if (StartPoint != null && EndPoint != null)
        {
            TSM.Beam Point = new TSM.Beam(StartPoint, EndPoint);
            Point.Name = Tools.PLUGIN_PART_POINT_NAME;
            Point.Class = Convert.ToString(_data.PointClass);
            Point.Finish = "";
            Point.PartNumber.Prefix = "";
            Point.PartNumber.StartNumber = -1;
            Point.AssemblyNumber.Prefix = "";
            Point.AssemblyNumber.StartNumber = -1;
            Point.Material.MaterialString = Tools.PLUGIN_PART_POINT_MATERIAL;

            Point.Profile.ProfileString = UsedProfileString + SizeString;
            Point.Position.Depth = TSM.Position.DepthEnum.MIDDLE;
            if (!Point.Insert())
                throw new Exception("Error: Fail to insert new layout point");
            Point.SetUserProperty(Tools.PLUGIN_PROPERTY_GROUP_NAME, _data.PointGroupName);
            Point.SetUserProperty(Tools.PLUGIN_PROPERTY_PointLabel, _data.PointLabel);
            Point.SetUserProperty(Tools.PLUGIN_PROPERTY_PointDescription, _data.PointDescription);
            if (Point.SetUserProperty(Tools.PLUGIN_UDA_EXISTING_STATUS, 0))
            {
                if (!Point.Modify())
                    MessageBox.Show("Error: Fail to write existing member UDA to line object");
            }
        }
```

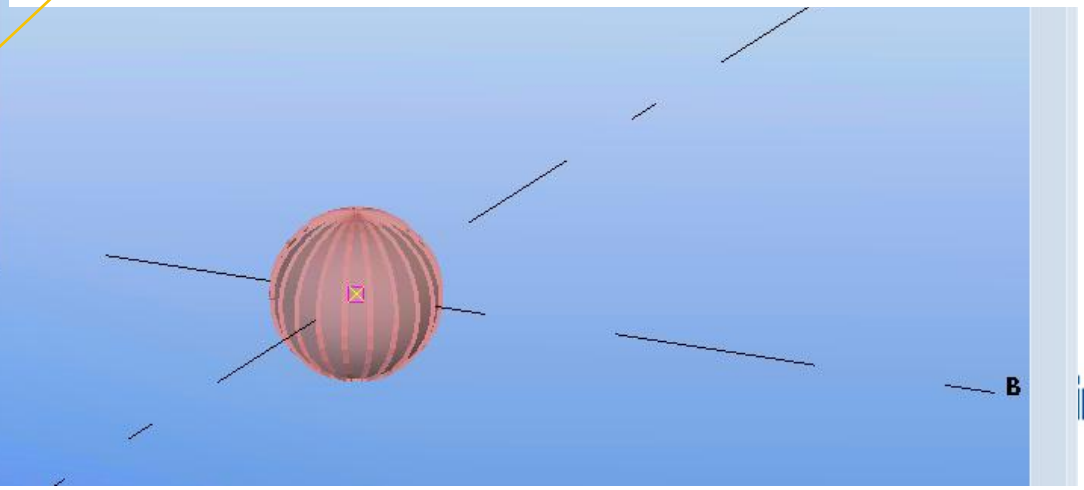
# Layout Point



# Layout Point

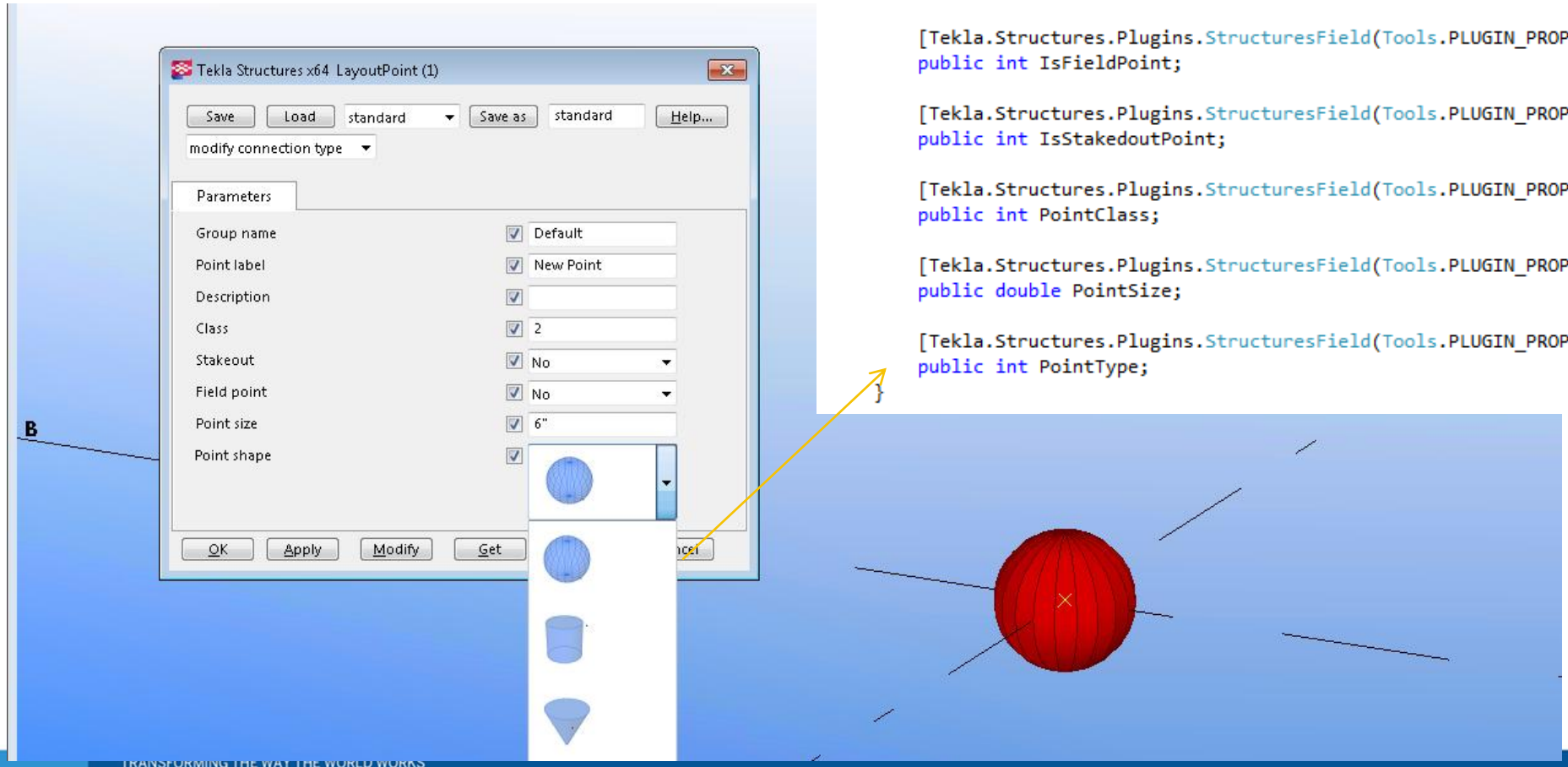



```
namespace Tekla.Structures.Layout
{
    public class LayoutPointData
    {
        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointGroupName)]
        public string PointGroupName;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointLabel)]
        public string PointLabel;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointDescription)]
        public string PointDescription;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_IsFieldPoint)]
        public int IsFieldPoint;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_IsStakedoutPoint)]
        public int IsStakedoutPoint;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointClass)]
        public int PointClass;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointSize)]
        public double PointSize;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointType)]
        public int PointType;
    }
```

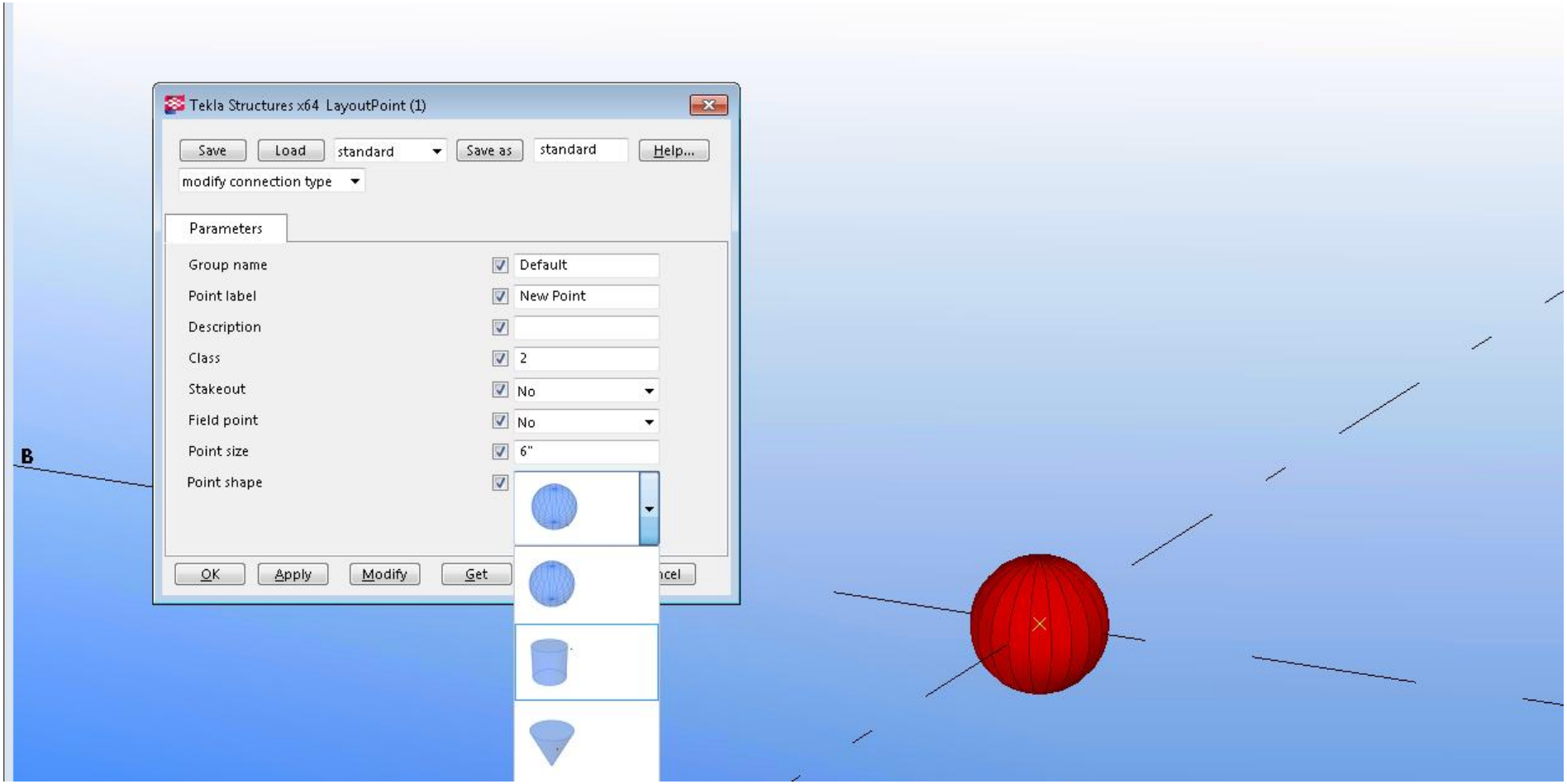


# Layout Point

```
namespace Tekla.Structures.Layout
{
    public class LayoutPointData
    {
        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointGroupName)]
        public string PointGroupName;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointLabel)]
        public string PointLabel;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointDescription)]
        public string PointDescription;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_IsFieldPoint)]
        public int IsFieldPoint;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_IsStakedoutPoint)]
        public int IsStakedoutPoint;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointClass)]
        public int PointClass;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointSize)]
        public double PointSize;

        [Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointType)]
        public int PointType;
    }
```

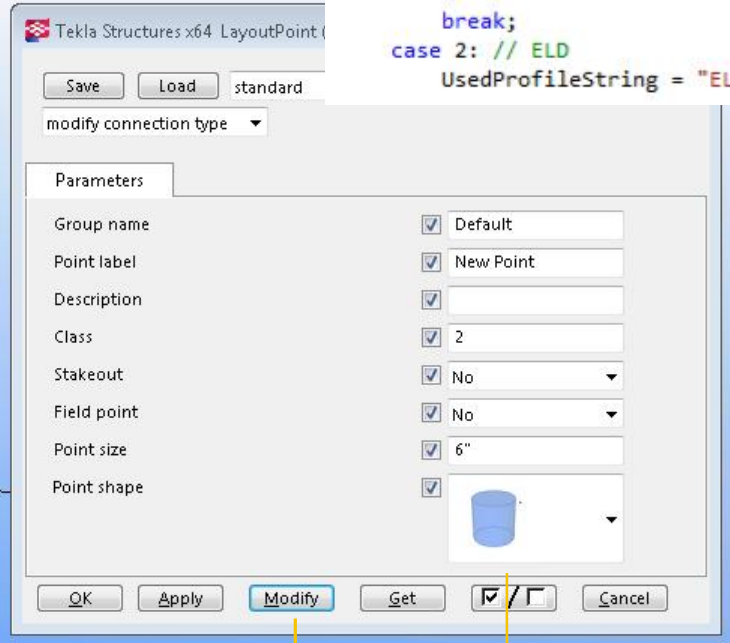


# Layout Point

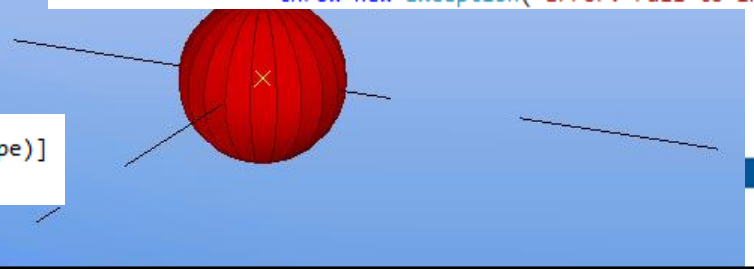


# Layout Point

```
public override bool Run(List<InputDefinition> input)
{
    try
    {
        // Manage Profile
        switch (_data.PointType)
        {
            case 1: // ROD

                StartPoint = new Tekla.Structures.Geometry3d.Point(PickPoint.X, PickPoint.Y, PickPoint.Z - size);
                EndPoint = new Tekla.Structures.Geometry3d.Point(PickPoint.X, PickPoint.Y, PickPoint.Z + size);
                UsedProfileString = "ROD";

                //Set Size String
                SizeString = _data.PointSize.ToString();

                break;
            case 2: // ELD
                UsedProfileString = "ELD";
```



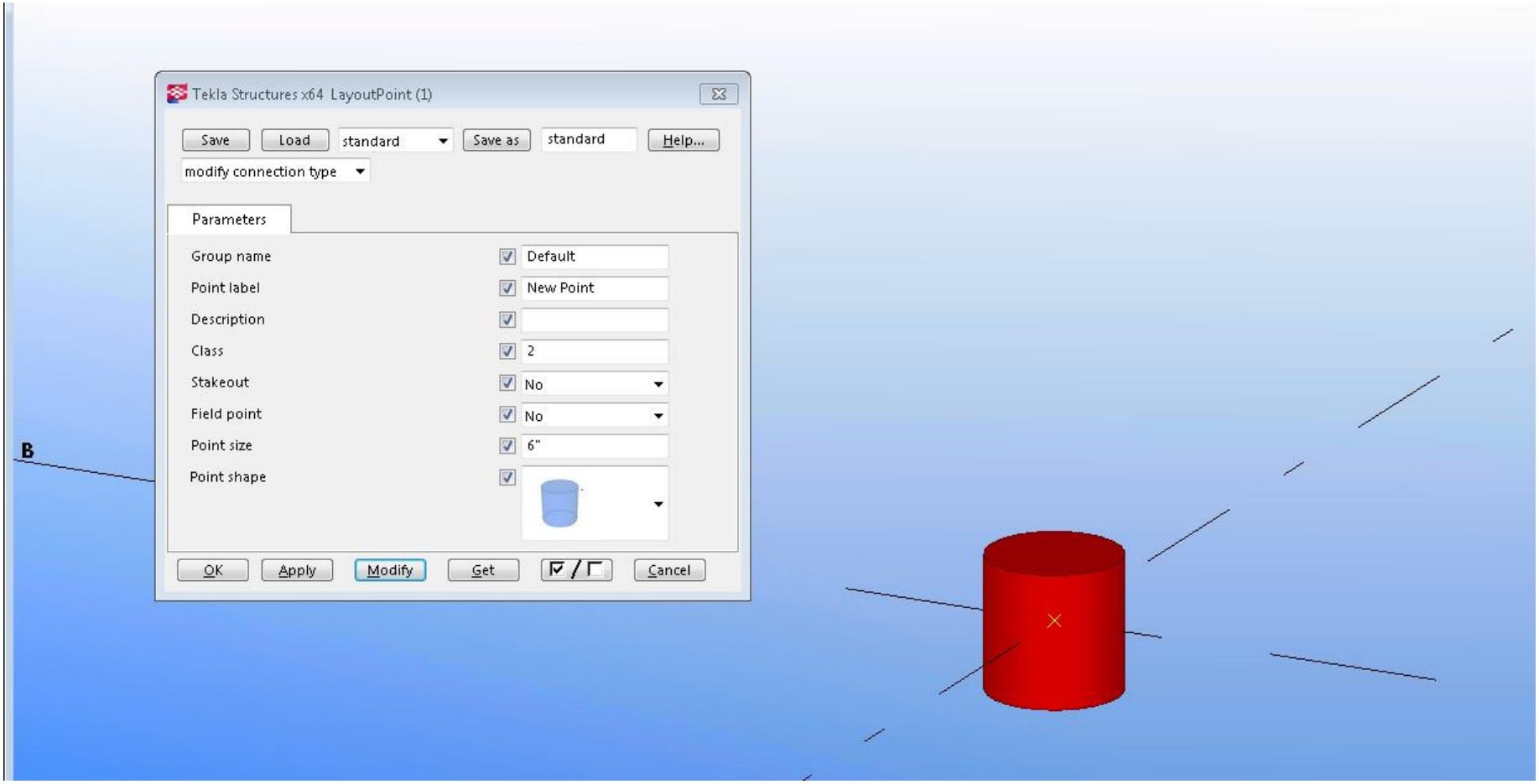

```
        if (StartPoint != null && EndPoint != null)
        {
            TSM.Beam Point = new TSM.Beam(StartPoint, EndPoint);
            Point.Name = Tools.PLUGIN_PART_POINT_NAME;
            Point.Class = Convert.ToString(_data.PointClass);
            Point.Finish = "";
            Point.PartNumber.Prefix = "";
            Point.PartNumber.StartNumber = -1;
            Point.AssemblyNumber.Prefix = "";
            Point.AssemblyNumber.StartNumber = -1;
            Point.Material.MaterialString = Tools.PLUGIN_PART_POINT_MATERIAL;

            Point.Profile.ProfileString = UsedProfileString + SizeString;
            Point.Position.Depth = TSM.Position.DepthEnum.MIDDLE;
            if (!Point.Insert())
                throw new Exception("Error: Fail to insert new layout point");
```



```
[Tekla.Structures.Plugins.StructuresField(Tools.PLUGIN_PROPERTY_PointType)]
public int PointType;
```

Trimble.

# Layout Point



# Layout Point – Case summary

- Plugin input definition
  - DefineInput() in PluginBase and attributes in ConnectionBase
- Plugin dialog attributes
  - [StructuresField(”param”)]
- Executing and modifying a plugin
  - Run()
  - Input definitions
  - Dialog parameters

# Layout Point – Case summary

- Utilizes Tekla Plugins and Tekla Model APIs
- Only a few hundred lines of code
- Very simple yet powerful plugin
- Fits a specific need

# Plugin Basics

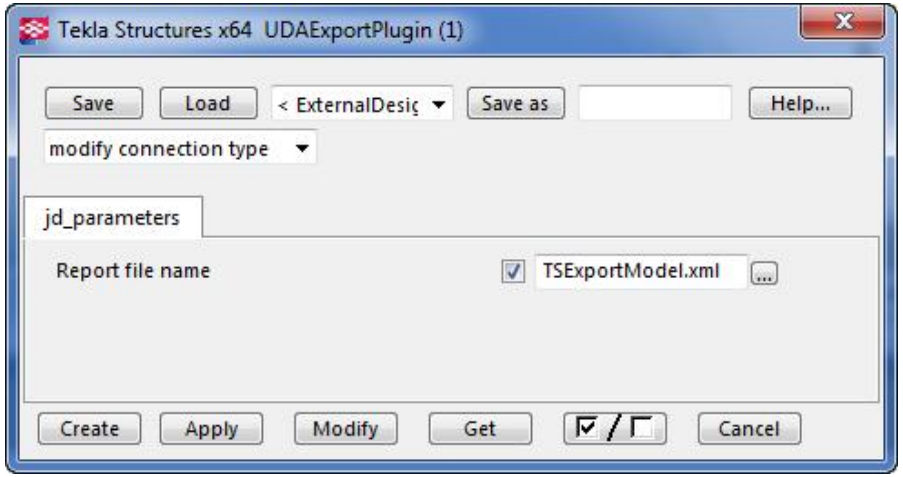
Non-dependent plug-ins

# Objective

- Understand the basics of Non-dependent plug-ins
  - Definition
  - How to execute
- Understand where to use of Non-dependent plug-ins
- Difference between Applications and Plug-ins.

# How non-dependent plug-ins work?

- Plug-in dialog is opened from component catalog and executed from dialog
- Values can be applied from dialog normally
- Plug-in instance is deleted after execution
- Plug-in dialog cannot be opened from created objects
- Created objects are saved to database as native objects



# Where to use non-dependent plug-ins?

- Can be used in tools where a lot of model information needs to be fetched or set
- Data is transferred in-process through API, remoting not used
  - 20x faster execution when  compared to applications
- Typically used in
  - import and export functionality
  - Status-checking tools
- Also useful in tools where model information needs to be consistent
  - Tekla Structures UI is “frozen” during plug-in execution
  - External applications cannot modify data during execution

# Setup

- Virtually the same as normal model plug-in
- Has Run and define input override
- Receives data the same way from user interface

```
[Plugin("PourStopApplicator")]
[PluginUserInterface(PourStopApplicatorInp.MainInpDefinition)]
[InputObjectDependency(InputObjectDependency.NOT_DEPENDENT)]
public class PourStopApplicator : PluginBase
```

```
new Model().CommitChanges();
```

# Setup

- Input not mandatory
  - Empty list returned from DefineInput()

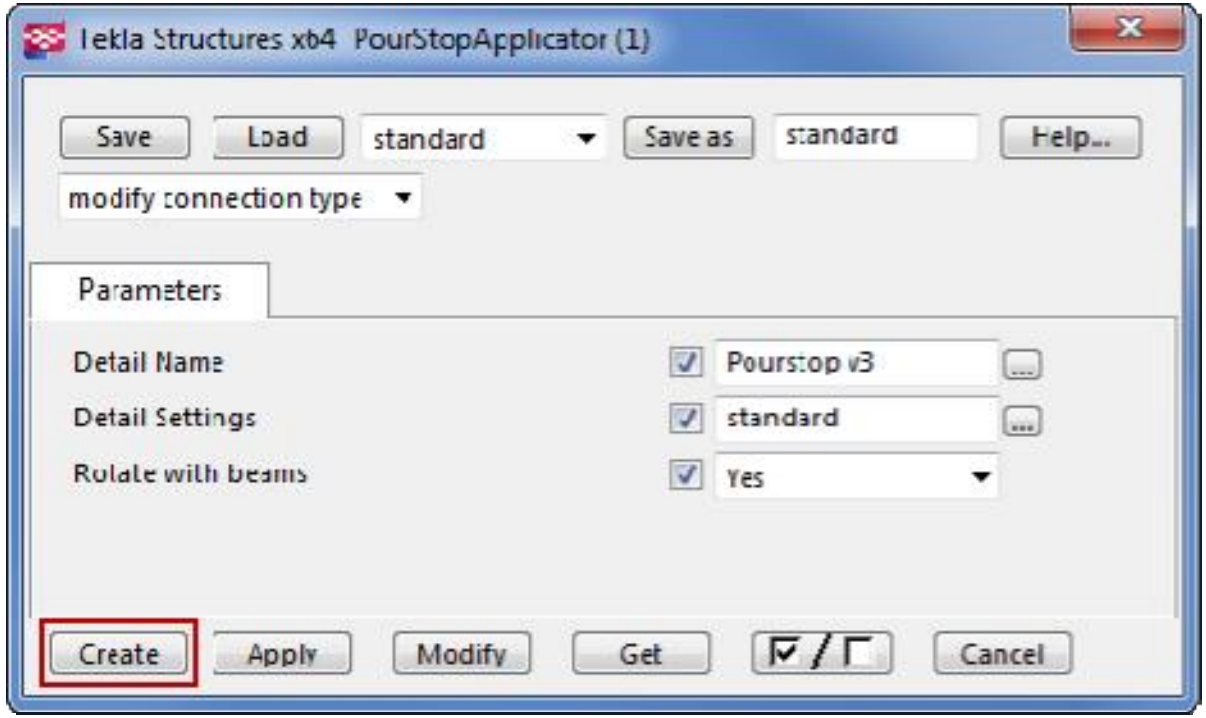
```
public override List<InputDefinition> DefineInput()
{
    return new List<InputDefinition>();
}
```

- Forms and inp can be used for dialog definition
  - Modify()-method executes plug-in if Forms is used

# Example Non-Dependent Plug-in

# Use Cases

- Applicator tool
- Run tasks very quickly



# Define Input

```
public override List<InputDefinition> DefineInput()
{
    var beamPicker = new Picker();
    var inputList = new List<InputDefinition>();
    try
    {
        //Get input from user
        var pickedBeamIdentifiers = new ArrayList();
        var pickedBeamEnum = beamPicker.PickObjects(Picker.PickObjectsEnum.PICK_N_PARTS, PickBeamsPrompt);
        while (pickedBeamEnum.MoveNext())
        {
            var bm = pickedBeamEnum.Current as Beam;
            if (bm == null) continue;
            pickedBeamIdentifiers.Add(bm.Identifier);
        }

        //Add inputs to InputDefinition list and return
        inputList.Add(new InputDefinition(pickedBeamIdentifiers));
        return inputList;
    }
    catch (Exception ex)
    {
        //Catch common user interrupt exception from Tekla
        if (ex.Message.Contains("interrupt")) return null;
        throw;
    }
}
```

# Run

```
public override bool Run(List<InputDefinition> input)
{
    //Check data for default values
    Data.CheckDefaults(null);
    _lastBeam = null;

    //Get data from input and run code to insert details
    var counter = 0;
    var pickedBeamIdentifiers = (ArrayList)(input[0]).GetInput();
    foreach (var identObj in pickedBeamIdentifiers)
    {
        var identifier = identObj as Identifier;
        if (identifier == null) continue;
        var bm = new Model().SelectModelObject(identifier) as Beam;
        if (bm == null) continue;
        if (CreatePourStop(bm, Data)) counter++;
    }
    new Model().CommitChanges();
    return counter > 0;
}
```

# Data Storage Class

```
public class PourStopApplicatorData
{
    [StructuresField("DetailNumber")]
    public int DetailNumber;

    [StructuresField("detail_name")]
    public string DetailName;

    [StructuresField("detail_attrfile")]
    public string DetailSettings;

    [StructuresField("RotateSide")]
    public int RotateSide;

    public void CheckDefaults(PluginBase plugin)
    {
        if (plugin == null) return;
        if (plugin.IsDefaultValue(DetailNumber)) DetailNumber = -100;
        if (plugin.IsDefaultValue(DetailName)) DetailName = "Pourstop v3";
        if (plugin.IsDefaultValue(DetailSettings)) DetailSettings = "standard";
        if (plugin.IsDefaultValue(RotateSide)) RotateSide = 0;
    }
}
```

# Insert Existing Detail

- Use Detail, Custom Part, Seam, Connection, and Component classes to insert existing custom components, system components, and plug-ins
- Insert new instance of your plug-ins or Tekla plug-ins

```
//Create new instance of detail in memory with settings needed
var detail = new Detail { Name = uiData.DetailName, Number = GetNumber(uiData) };
detail.LoadAttributesFromFile(uiData.DetailSettings);
detail.SetPrimaryObject(tBeam);
//detail.AutoDirectionType = AutoDirectionTypeEnum.AUTODIR_BASIC;
detail.DetailType = DetailTypeEnum.END;

//Set attributes for detail to apply correctly
SetDirectionAttribute(tBeam, uiData, detail);

//Insert detail into model
return detail.Insert();
```

# Drawing Plugins Types

```
/// <summary>
/// The plug-in is never updated.
/// Plug-ins are executed from the plug-in dialog instead of the component catalog.
/// The created objects do not have any relation to the plug-in anymore.
/// The plug-in dialog cannot be opened from the created objects.
/// </summary>
CREATE_ONLY = 0,

/// <summary>
/// The plug-in is updated when the input is a point and the point is moved or when the input is an object and the object
/// changes. The plug-in is executed when the input is an object and its properties are changed in the drawing editor.
/// This mode is the default which is used if the update mode is not defined in the plug-in source.
/// </summary>
INPUT_CHANGED = 1,

/// <summary>
/// The plug-in is updated also when a drawing is opened.
/// The plug-in is executed when the input is changed or during drawing opening.
/// </summary>
DRAWING_OPENED = 2
```

# Tekla Open API

## Dialog basics

§ Tekla Open API: Plug-ins & Dialogs

# Objective

- Understand the basics of Dialogs
  - Attribute binding.
  - Use of Distance/DistanceList
- Understand the use of UIControls
  - How the controls work.
- Difference between Applications and Plug-ins.
- Difference between INP and Forms dialogs.

# Dialog Types

- FormBase class provides localization, unit conversion and data storage among other things. Abstract class: should not be directly inherited
- ApplicationFormBase: Defines forms for Applications.
- PluginFormBase: Defines forms for Plug-ins.
- Inheriting from this Forms enables structuresExtender which adds extra properties to the controls.

# Attribute Binding

- Attributes: pass data from the Form into the Plug-in.
- StructuresData: defines all the data (Attributes) that can be pass from the UI.
- Check boxes: filter attributes.

```
public class StructuresData
{
        [TSPlugins.StructuresField("height")]
        public double height;
```

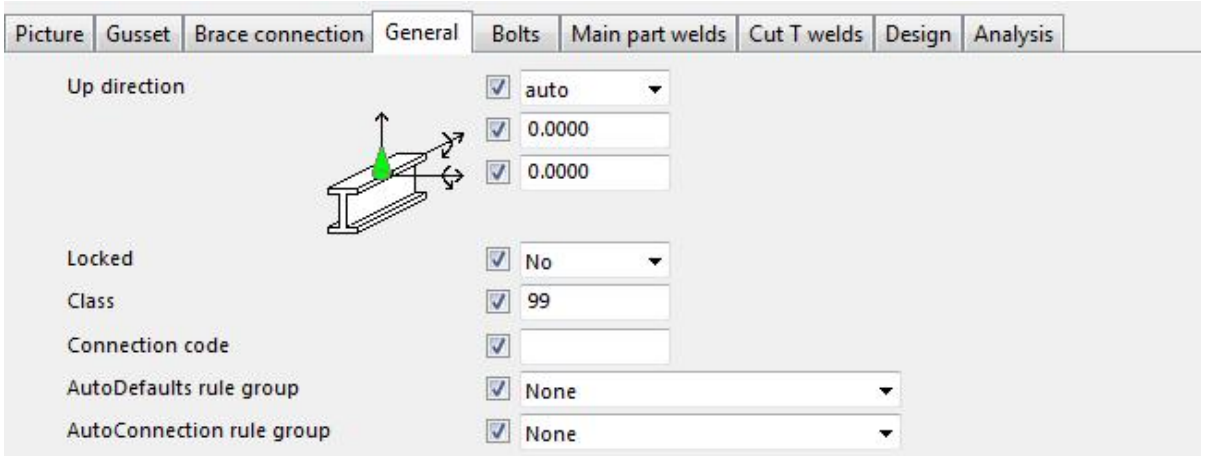
| Tekla Structures | |
|---|---|
| AttributeName | **height** |
| AttributeTypeName | **Distance** |
| BindPropertyName | |
| IsFilter | False |

| Tekla Structures | |
|---|---|
| AttributeName | **height** |
| AttributeTypeName | |
| BindPropertyName | |
| IsFilter | **True** |

- BindPropertyName: bind to other property than the default one of the control (e.g: combobox SelectedIndex)

# INP notes

- General tab in Connections
- Access to Component Catalog



# Avoid deadlocks in Plug-ins

- Picker: must be launched in a separate thread when called in the Run or in the Form.
- MessageBox: stops the Plug-in until clicked. In Form plugins must be launched in a separate thread when called in the Run.

# ApplicationFormBase vs PluginFormBase

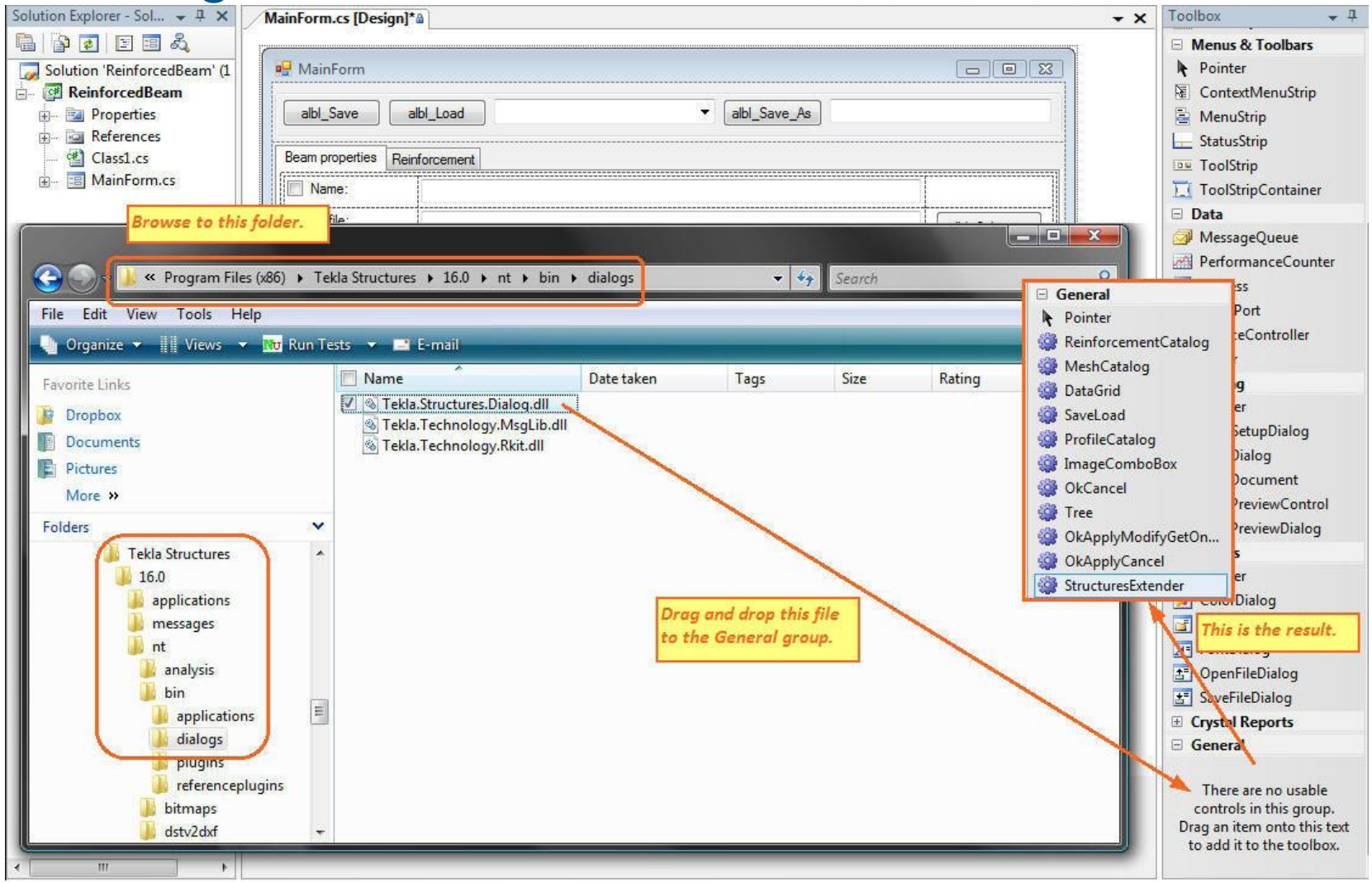
- ApplicationFormBase implements less methods: no need for Modify, Get,…
- Applications have to call base.InitializeForm() to initialize the data storage.

```
public partial class Form1 : ApplicationFormBase
{
    public Form1()
    {
        InitializeComponent();
        base.InitializeForm();
    }
}
```

# UIControls

- Contains custom controls to ease the implementation of dialogs.
- Can be found in Tekla.Structures.Dialog.UIControls

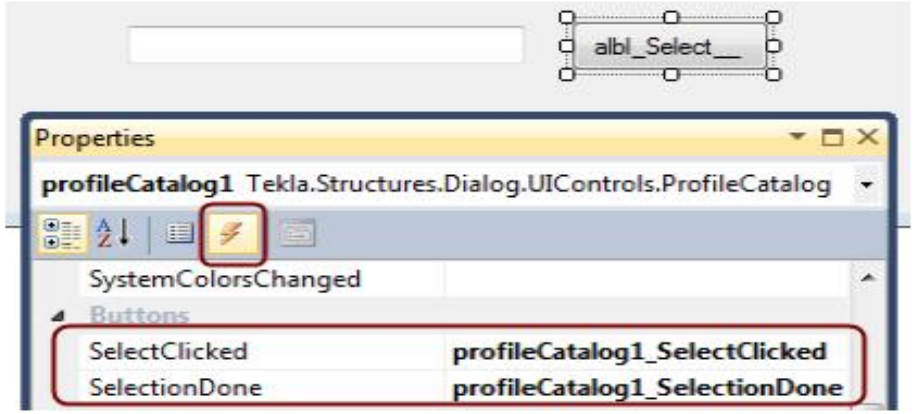
# Adding UIControls to the Toolbox

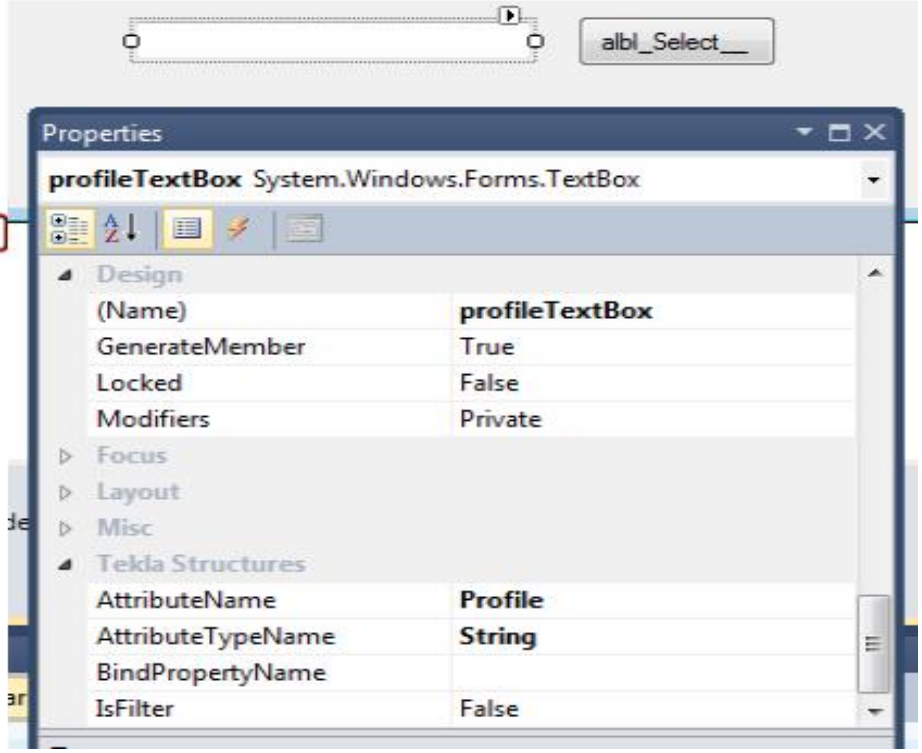


# UIControls: Catalogs

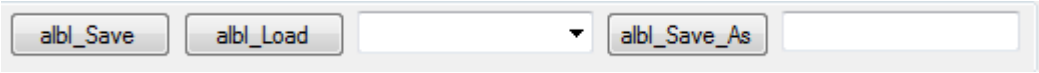
§ Access to catalogs: Bolt, Material, Mesh, Profile and Reinforcement.

```
private void profileCatalog1_SelectClicked(object sender, EventArgs e)
{
    profileCatalog1.SelectedProfile = profileTextBox.Text;
}

private void profileCatalog1_SelectionDone(object sender, EventArgs e)
{
    this.SetAttributeValue(profileTextBox, profileCatalog1.SelectedProfile);
}
```



# UIControls: SaveLoad




- SaveLoad: represents a  Save-Load-Save As group of controls including the functionality.
- No need to set up, just drag and drop into the Form.
- Only controls bound to an attribute and user attributes (not bound to controls) will be saved.
- The attribute file is saved in the <model>\attributes folder.
- The file extension is <namespace>.<FormName>.xml. E.g. standard.Splice.Attributes.xml

# Notes

- Beware of typos in the Attributes names, they have to be the same in the control and in the StructuresData.
- Attribute name cannot be more than 18 characters
- String Attributes value cannot be more than 79 characters.

```
public class StructuresData
{
        [TSPlugins.StructuresField("height")]
        public double height;
```

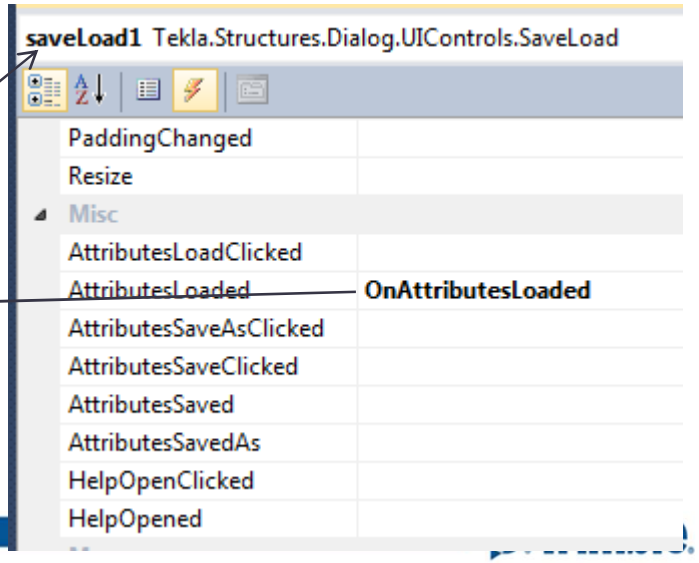
| Tekla Structures | |
|---|---|
| AttributeName | **height** |
| AttributeTypeName | **Distance** |
| BindPropertyName | |
| IsFilter | False |

| Tekla Structures | |
|---|---|
| AttributeName | **height** |
| AttributeTypeName | |
| BindPropertyName | |
| IsFilter | **True** |

# SaveLoad and DataGrid

- SaveLoad + DataGrid: not direct support. Needs to use User Attributes.
- You can find more information in the forum:
  - https://extranet.tekla.com/FORUM/default.aspx?g=posts&t=4193
  - https://extranet.tekla.com/FORUM/default.aspx?g=posts&m=17938#post1793


```
private void OnAttributesLoaded(object sender, EventArgs e)
{
    GetDataGridViewValues();
}
```



```
        private void GetDataGridViewValues()
        {
            int Count = Math.Min(this.GetAttributeValue<Integer>(ColumnData.RowCount),
ColumnData.MAXROWS);
            dataGridView1.Rows.Clear();
            for (int i = 0; i < Count; i++)
            {
                dataGridView1.Rows.Add();
                for (int j = 0; j < ColumnData.ColumnTypes.Length; j++)
                {
                    if (ColumnData.ColumnTypes[j] == typeof(TSD.String))
                    {
                        dataGridView1.Rows[i].Cells[j].Value =
this.GetAttributeValue<TSD.String>(dataGridView1.Columns[j].Name + i.ToString());
                    }
                    if (ColumnData.ColumnTypes[j] == typeof(Distance))
                    {
                        dataGridView1.Rows[i].Cells[j].Value =
this.GetAttributeValue<Distance>(dataGridView1.Columns[j].Name + i.ToString()).ToString();
                    }
                }
            }
        }

        private void SetDataGridViewValues()
        {
            int Count = Math.Min(dataGridView1.Rows.Count, ColumnData.MAXROWS);
            this.SetAttributeValue(ColumnData.RowCount, new Integer(Count));
            for (int i = 0; i < Count; i++)
            {
                for (int j = 0; j < ColumnData.ColumnTypes.Length; j++)
                {
                    if (ColumnData.ColumnTypes[j] == typeof(TSD.String))
                    {
                        TSD.String value = new
TSD.String(dataGridView1.Rows[i].Cells[j].Value.ToString());
                        this.SetAttributeValue(dataGridView1.Columns[j].Name + i.ToString(), value);
                    }
                    if (ColumnData.ColumnTypes[j] == typeof(Distance))
                    {
                        Distance result;
                        Distance.TryParse(dataGridView1.Rows[i].Cells[j].Value.ToString(), out result);
                        this.SetAttributeValue(dataGridView1.Columns[j].Name + i.ToString(), result);
                    }
                }
            }
        }
```

```
public override bool Run(List<InputDefinition> input)
{
    try
    {
    ...

    // get values from grid
    Component myComponent = new Component();
    myComponent.Identifier = this.Identifier;

    int rowCount = 0;
    myComponent.GetAttribute(ColumnData.RowCount, ref rowCount);

    Hashtable strTable = new Hashtable();
    myComponent.GetStringUserProperties(ref strTable);
    Hashtable dblTable = new Hashtable();
    myComponent.GetDoubleUserProperties(ref dblTable);

    for( int i = 0; i<rowCount; i++ )
    {
        TSModel.Beam column = new TSModel.Beam();
        column.StartPoint = new TSGeometry.Point(startPoint);
        column.EndPoint = new TSGeometry.Point(endPoint);
        column.Profile.ProfileString = strTable[ColumnData.ColumnNames[0] + i.ToString()].ToString();
        column.Material.MaterialString = strTable[ColumnData.ColumnNames[1] + i.ToString()].ToString();
        column.StartPoint.X += (double)dblTable[ColumnData.ColumnNames[2] + i.ToString()];
        column.StartPoint.Y += (double)dblTable[ColumnData.ColumnNames[3] + i.ToString()];
        column.StartPoint.Z += (double)dblTable[ColumnData.ColumnNames[4] + i.ToString()];
        column.EndPoint.X += (double)dblTable[ColumnData.ColumnNames[2] + i.ToString()];
        column.EndPoint.Y += (double)dblTable[ColumnData.ColumnNames[3] + i.ToString()];
        column.EndPoint.Z += (double)dblTable[ColumnData.ColumnNames[4] + i.ToString()];

        // Insert the beam in the model
        column.Insert();
    }
```

# DistanceList

- Control properties set as DistanceList.
- StructuresData Attribute set as string.
- Parse the Attribute into a DistanceList.
- Loop through the values in the distance list when using them.

| Tekla Structures | |
|---|---|
| AttributeName | **Spacing** |
| AttributeTypeName | **DistanceList** |
| BindPropertyName | |
| IsFilter | False |

```
public class StructuresData
{
    ...

    [StructuresField("Spacing")]
    public string Spacing; // use Datatype.DistanceList.Parse to get array of doubles
in source
}
}
```

```
DistanceList Spacing = DistanceList.Parse(this.Data.Spacing, CultureInfo.CurrentCulture,
Distance.UnitType.Millimeter);
```

```
foreach (var distance in this.Spacing)
{
    orientation.GetNormal();
    orientation.Normalize(distance.Millimeters);
    point1 = point1 + orientation;
    endpoint = point1 + new Point(0, 0, this.Height);
    this.CreateBeam(point1, endpoint);
}
```

# Thank You