Tekla Open API

Developer's Guide

Tekla Structures 2016

Contents

Introduction

What is Tekla Open API?

Interface is a common boundary between two entities.

Tekla Structures *user interface* enables human users to communicate with the software.

Tekla's Open API, *Application Programming Interface*, facilitates interaction between Tekla Structures and other software.

The Tekla Open API, also known as the .NET API, provides an interface for third party applications to interact with model and drawing objects in Tekla Structures. It also allows the creation of plug-ins.

The Tekla Open API is developed on the Microsoft .NET Framework version 4.0 Full Profile.

Where to utilize the Tekla Open API?

Tekla Open API can be utilized with variety of tools. Tekla Open API enables users and vendors to for example:

Automate routine tasks

By recording and running user interface actions it is possible to automate routine tasks such as creating daily reports.

With the Tekla Open API you can automate for example the creation of modeling and drawing objects. These creation tools are typically frequently needed in projects and can for instance:

- Create basic structures to model such as type base hall
- Insert typical AutoCAD details to drawings based on drawing information
- Automate creation of dimensions for GA drawing based on model information

Integrate Tekla Structures into your own process, workflow, and software

Tekla Open API can be utilized in integration of Tekla Structures to other software. Information can be transferred between Tekla Structures model and drawing database and other software used in your office such as:

- Office tools
- A&D
- MIS & ERP

With the API you can for instance get full geometric information for CNC.

Develop additional functionality to extend and enhance Tekla Structures

It is possible to create tools which will add functionality or information to Tekla Structures through the API such as:

- RFI (Request for Information) management
- New connection creator between parts
- Model check and correction tools
- Erection and site planning tools
- Calculating externally values for reports
- Recording and running recorded UI macros.

Where to find support?

Tekla Online Services is the Tekla's channel of supporting .NET developers. From there you can find the Tekla Open API page (see http://www.tekla.com/openAPI) with links downloads section as well as the Open API discussion forum.

Start-up package

The Start-up package, that can be downloaded from the Tekla Warehouse, is a good starting point for learning and developing your first application or plug-in.

Open API discussion forum

Support in the forum is based on C#. It contains up-to-date information and answers to specific questions about using the API. Besides that forum collects also feature requests and bugs.

Before posting your question to the forum it might be useful to use the Search functionality provided by the forum – quite often your question have been answered already. And please do not hesitate to answer yourself if you can be of help to others!

Note that the forum is not meant for general questions about .NET and programming, or Microsoft's or other 3rd party application's APIs.

About Assemblies

Introduction

Assemblies and reference manual can be found in the folders “nt\bin\plugins”, “nt\bin\dialogs” and “nt\help”, respectively, in each version of Tekla Structures.

All assemblies except Tekla.Structures.Plugins and Tekla.Structures.CustomPropertyPlugin are registered to GAC (Global Assembly Cache).

Common principles

All object classes in the API have common methods for manipulating objects such as:

- Select() for object selection from database or from interface
- Insert() for creating new object instance
- Modify() for modifying object in database or interface
- Delete() for deleting object from database or interface

All object classes in API have been designed with following principles:

- Object initialization is similar to object creation from user interface of Tekla Structures. As example Beam is initialized by giving two positions in constructor-method.
- Field naming in object classes follows dialog field names (in English language) as much as possible.

Tekla.Structures.dll

This assembly contains some basic common types shared by model and drawings.

Tekla.Structures.Model.dll

Model assembly consists of classes and methods used when connecting and manipulating objects in Tekla Structures model database:

- Model object classes such as Beam, ContourPlate, BoltArray, Load, Component, etc.
- Methods to manipulate model objects, including basic object operations like copy, move, combine and split
- Methods to enumerate model objects
- Methods such as picking (dynamic selection) and selection of model views
- Analysis classes for accessing the analysis results database
- Classes related to clash checking

Besides Tekla.Structures.Model namespace this assembly contains sub-namespaces for ClashChecker, Collaboration, History, Operations and UI.

Tekla.Structures.Drawings.dll

Drawings assemblies are recommended to be used to edit the drawings, not for drawing creation.

This assembly consists of classes and methods used when connecting and manipulating objects in Tekla Structures drawing database:

- Drawing object classes such as Drawing, View, Part, Text, Line.
- Methods to manipulate and enumerate drawing objects.
- Methods such as picking (dynamic selection) and selection for drawing views.

This assembly contains four namespaces: Tekla.Structures.Drawings, Tekla.Structures.Drawings.UI, Tekla.Structures.Drawings.Automation, and Tekla.Structures.Drawings.Tools.

Tekla.Structures.Plugins.dll

Plugins assembly consists of classes and methods used when defining .NET component tools (plug-ins) to Tekla Structures. The functionality of a plug-in is similar to a generic component tool created using Developer Kit. The definition contains:

- User interface
- Input management
- Database structure
- Execution

Tekla.Structures.Analysis.dll

This assembly includes the classes to be used for accessing Analysis & Design information in Tekla Structures.

Tekla.Structures.Catalogs.dll

Catalogs contain information on available profiles, bolts, materials, rebars, meshes and printers. For example, the bolt catalog contains a library of standard bolts and bolt assemblies used in structural steelwork.

With this assembly you can for example:

- Enumerate the profiles from the profile catalog.
- Collect the profiles' information (name or prefix, type, subtype, parameter string and parameters).
- Select a library profile item by its profile name.

Tekla.Structures.Datatype.dll

The provided unit types are used to pass data to and from Tekla Structures and between other Tekla Structures assemblies.

The following unit types are provided:

- Distance
- DistanceList
- Boolean
- Double
- Integer
- String

Conversions are supported only with distance unit.

Tekla.Structures.Dialog.dll

This assembly:

- Enables dialog creation for Tekla Structures plug-ins
- Enables a data connection to Tekla Structures plug-ins
- Enables localization of dialogs
- Enables support for Tekla Structures data types and conversions

- Enables default storing of values

The Tekla.Structures.Dialog.UIControls namespace contains dialog templates and custom controls. For more information see chapter “User interface controls”.

NOTE: This assembly is located in “nt\bin\dialogs” folder instead of “nt\bin\plugins”.

Tekla.Structures.CustomPropertyPlugin.dll

Tekla.Structures.CustomPropertyPlugin assembly consists of classes and metadata definitions which are needed for new .NET custom properorty (plug-ins) to Tekla Structures. Custom properties are used in external template property value calculations for reports, drawings and filtering.

The Tekla.Structures.CustomPropertyPlugin assembly provides `ICustomPropertyPlugin` interface which must be implemented when a new custom property is defined (see topic Custom Properties later in this document).

Applications

Introduction

Application can be for example

- .NET application, which can be Windows Forms or console applications
- COM application utilizing COM technology
- VBA macro utilizing COM-technology i.e. like in Microsoft Office

The simplest way of using the Tekla Open API is a .NET application.

Application is executed in a separated process and cannot be started from Tekla Structures (unless embedded to a macro).

Getting started

To start using the API you need to add references to the API assemblies and directives to the namespaces in the source code. Please refer the Self Learning Exercises for step-by-step instructions.

A .NET application that utilizes the API is usually run after Tekla Structures has been started and a model opened. Because applications are not started together with Tekla Structures they cannot thus expect to have Tekla Structures running upon execution.

Before interacting with the model, a "handle" to the model must be created; for example:

```
// First we must connect to the model by creating a "handle"
Model model = new TSM.Model();
// Now we check that the model connection succeeded
if (model.GetConnectionStatus())
{
  ...
```

For drawings, the following creates a "handle" to the drawings object:

```
DrawingHandler CurrentDrawingHandler = new DrawingHandler();
```

Once a “handle” is obtained, it is possible to insert, select, modify, delete or query model and drawings objects in Tekla Structures.

Example: ask user to pick objects

It is possible to prompt the user to select object(s) in Tekla Structures; this is done by using the Picker class.

Picker class for model objects:

```
Picker Picker = new Picker();
Point  p = null;
try
{
    p = Picker.PickPoint();
}
catch (Exception e)
{
    // User cancels selection (interrupt)
}
```

Picker class for drawings objects:

```
DrawingHandler drawingHandler = new DrawingHandler();

Picker picker = drawingHandler.GetPicker();

Point point = null;
ViewBase view = null;

try
{
    picker.PickPoint("Pick_point", out point, out view);
}
catch (PickerInterruptedException)
{
    // User cancels selection (interrupt)
}
```

Example: go through objects in the model or a drawing

Another useful class is ModelObjectEnumerator, which provides the means to iterate through model objects in the current model; for drawings, the class is DrawingObjectEnumerator.

An example of model objects enumeration:

```
ModelObjectEnumerator Enum =
    Model.GetModelObjectSelector().GetAllObjects();

while(Enum.MoveNext())
{
```

```
    Beam B = Enum.Current as Beam;
    if(B != null)
    {
        Solid Solid = B.GetSolid();
    }
}
```

An example of drawings objects enumeration:

```
DrawingObjectEnumerator AllObjects =
    CurrentDrawing.GetSheet().GetAllObjects();

while(AllObjects.MoveNext())
{
    if(AllObjects.Current is Line)
    {
        AllObjects.Current.Delete();
    }
}
```

ApplicationFormBase

ApplicationFormBase adds the following support:

- Enables Tekla Structures data types and conversions
- Enabling default storing of values
- Enables multiple language support

The applications that want to make full use of *ApplicationFormBase* should inherit from it. However this is not needed to when only making use of some of its features.

Applications inheriting *ApplicationFormBase* **must** call *InitializeForm()*. This must be called after the InitializeComponent() method of the Form:

```
using Tekla.Structures.Dialog;

public class MainForm : ApplicationFormBase
{
    MainForm()
    {
        InitializeComponent();
         InitializeForm();
    }
}
```

Attributes can be associated to UI controls. There is more information how to do this in chapter Data connections for Plug-ins.

The following code example shows one way to initialize an application making use of a local XML file holding the translated strings (please read also chapter Localization of strings):

```
using Tekla.Structures.Dialog;
using Tekla.Structures.Datatype;
```

```
public class MainForm : ApplicationFormBase
{
    MainForm()
    {
        InitializeForm();

        if(GetConnectionStatus())
        {
            string messageFolder = null;

            Model.GetAdvancedOption("XS_MESSAGES", ref messageFolder);

            messageFolder = Path.Combine(messageFolder, @"DotAppsStrings");

            Dialogs.SetSettings(string.Empty);

            Localization.Language =
                (string) Settings.GetValue("language");

            Localization.LoadFile(Path.Combine(
                messageFolder, Application.ProductName + ".xml"));

            Localization.Localize(this);
        }
        else
        {
            MessageBox.Show("Tekla Structures is NOT running...");
        }
    }
}
```

The following code shows how to create a separate localization instance, enabling localization of several Forms (using the Localization field of Form1), without inheriting from *ApplicationFormBase* and connecting to Tekla Structures through a *Model* object:

```
using TeklaModel = Tekla.Structures.Model.Model;
using TeklaLocalization = Tekla.Structures.Dialog.Localization;

namespace LocalizationExample
{
    public partial class Form1 : Form
    {
        private static TeklaModel _model;
        private static TeklaLocalization _localization;

        public Form1()
        {
            InitializeComponent();
            Localization.Localize(this);
        }

        public TeklaModel Model
        {
            get
            {
                if (_model == null)
                {
                    _model = new TeklaModel();
                }
                return _model;
            }
        }

        public TeklaLocalization Localization
        {
            get
            {
                if (_localization == null)
                {
                    string language = string.Empty;
                    string languageFile = string.Empty;

                    Model.GetAdvancedOption("XS_LANGUAGE", ref language);
                    Model.GetAdvancedOption("XS_DIR", ref languageFile);

                    languageFile = Path.Combine(languageFile,
```

```
                        @"messages\DotNetDialogStrings.xml");

                    _localization = new TeklaLocalization();
                    _localization.LoadFile(languageFile);
                    _localization.Language = GetShortLanguage(language);
                }
                return _localization;
            }
        }

        private static string GetShortLanguage(string Language)
        {
            switch (Language)
            {
                case "ENGLISH":
                    return "enu";
                case "DUTCH":
                    return "nld";
                case "FRENCH":
                    return "fra";
                case "GERMAN":
                    return "deu";
                case "ITALIAN":
                    return "ita";
                case "SPANISH":
                    return "esp";
                case "JAPANESE":
                    return "jpn";
                case "CHINESE SIMPLIFIED":
                    return "chs";
                case "CHINESE TRADITIONAL":
                    return "cht";
                case "CZECH":
                    return "csy";
                case "PORTUGUESE BRAZILIAN":
                    return "ptb";
                case "HUNGARIAN":
                    return "hun";
                case "POLISH":
                    return "plk";
                case "RUSSIAN":
                    return "rus";
                default:
                    return "enu";
            }
        }
    }
```

You can load several localization files by calling method LoadFile for each of them. The translations are searched in the loading order and the first match is returned.

Plug-ins

Introduction

Plug-ins are component tools used to automate creation of model or drawing objects. Plug-ins are intelligent meaning they are modifiable and dependent of input objects.

Plug-ins are loaded inside the Tekla Structures process. Model plug-ins starts from the Component Catalog, while drawings plug-ins are started from a toolbar.

The Tekla.Structures.Plugins assembly provides the following classes for creating plug-ins:

- PluginBase class: an abstract base class for component tools
- PluginFormBase class: a base class for plug-ins that use Windows Forms for dialog design. Class adds the following support:
  - .NET dialogs
  - Data connection
  - Tekla Structures data types and conversions automatically
  - Multiple language support
  - Storing of default values
- ConnectionBase class: base class for defining Connections, Details and Seams. These types are more specialized and restricted by the input values than the ones derived from PluginBase.
- DrawingPluginBase class: an abstract base class for drawings plug-ins; the user-interface, the dialog, uses the same PluginFormBase above.

Basic steps for creating a plug-in

- Create a new project in Visual Studio: Visual C#, Windows, Class Library.
- Microsoft .NET Framework 4.0
- Add references to the right Tekla Structures assemblies in Visual Studio.

- More than one plug-ins can be created in the same dll. All the plug-ins in that dll has to be created under the same project.
- The dll containing the plug-in has to be copied into: \<Tekla Structures installation folder>\<version>\nt\bin\plugins. A sub-folder could also be created to store the dll.
- Post build event can be created to copy the dll on successful build to the correct location. See instructions how to specify build events from Visual Studio's Help.

Model plug-ins

Mandatory fields in the model plug-in

- Plug-in has to have a name
- StructuresData: gets the data from the user interface into the plug-in
- Constructor of the plug-in: copies the data from Tekla Structures
- DefineInput (public method): method that defines the input of the plug-in
- Run (public method): main method of the plug-in

Dialog design with Windows Forms

Windows Forms, which is a part of Microsoft's .NET Framework, is the recommended way to define the dialog. The other option is to use input file format (INP), for more information on that please see chapter "Dialog design with INP".

PluginFormBase class provides connectivity to Tekla Structures for plug-ins. Forms that inherit from *PluginFormBase* are automatically added to the Component Catalog, if the dll file is found under nt\bin\plugins folder. The dialog stays on top of the Tekla Structures main window and closes when TS closes.

PluginFormBase adds the following support:

- Enables .NET dialogs for Tekla Structures plug-ins.
- Enables a data connection to Tekla Structures plug-ins.
- Enables Tekla Structures data types and conversions.
- Enables multiple language support.
- Enabling default storing of values.

Class definition

The plug-in Form has to inherit from *PluginFormBase*:

```
using Tekla.Structures.Dialog;

public class MainForm : PluginFormBase
{
```

```
..
```

Naming the plug-in

The plug-in name will appear in the Component Catalog, and it has to be unique. You cannot have two plug-ins with the same name.

The user interface for the plug-in is defined in a metadata attribute of the plug-in class, *PluginUserInterface*.

```
using Tekla.Structures.Plugins;

[Plugin("DialogDemo")]
[PluginUserInterface(typeof(DialogDemo.MainForm))]
public class MainPlugin : PluginFormBase
{
 ..
```

If the plug-in is inherited from ConnectionPluginBase, the dialog class name must be the same as plug-in name:

```
    [Plugin("SpliceConnection")]
    [PluginUserInterface("SpliceConnection")]
```

However, plug-ins inherited from PluginBase or DrawingPluginBase do not have this limitation. There dialog class name does not have to be the same as plug-in name:

```
    [Plugin("FloorToolPlugin")]
    [PluginUserInterface("FloorTool.FloorToolForm")]
```

In both cases interface definition must refer to actual dialog class. In connections class just has to be outside namespace and class name must be the same as plug-in name.

Connection dialog class:

```
public partial class SpliceConnection : PluginFormBase
{
 ...
```

Plug-in and drawing plug-in dialog class:

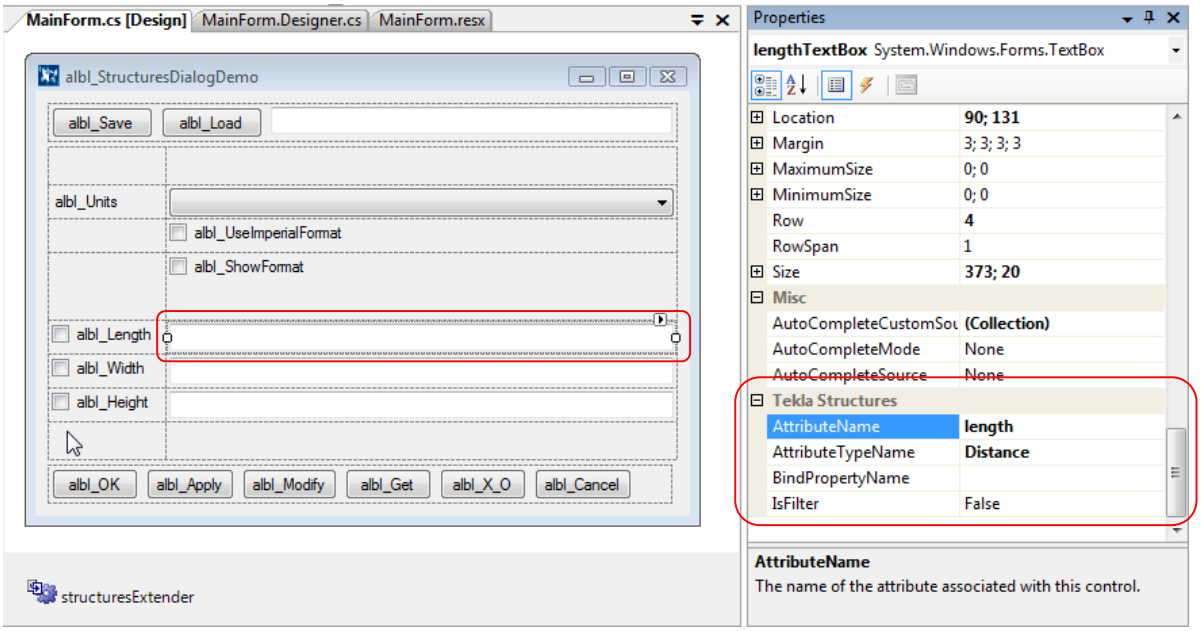
```
namespace FloorTool
{
   public partial class FloorToolForm :  PluginFormBase
   {
       ...
```

Note that PluginFormBase dialog class name cannot end with "_01" or similar.

Data connections

Plug-in dialogs save their data to Tekla Structures. The values are then passed to the corresponding plug-in when it is run.

To use a value entered in a dialog control in a plug-in, you need to set the Tekla Structures-specific attributes for each control. They can be found for each control in the Visual Studio Design view, in the Properties tab, under Tekla Structures (see picture below).



Plug-in dialog control properties.

The data is passed to the plug-in using a class defined by the developer of the plug-in. Its instance must be the only parameter for the plug-in's constructor and it must consist of public fields that have the Tekla.Structures.Plugins.StructuresField attribute. The attribute's parameter is where the binding is done to the dialog field. It must be the same as the name part of the *AttributeName* property for the field.

```
using Tekla.Structures.Plugins;

public class StructuresData
{
	[StructuresField("length")]
	public double length;
}

[Plugin("DialogDemo")]
[PluginUserInterface(typeof(DialogDemo.MainForm))]
public class DemoPlugin : PluginBase
{
	private StructuresData _data;
	public DialogDemo(StructuresData dialogData)
	{
		_data = dialogData;
	}

	public override bool Run(List<InputDefinition>
                                    input)
	{
		try
		{
			double Length = _data.length;
		}
		catch (Exception e)
		{
			[…]
		}
		return true;
	}
}
```

In the Run method (for example), of the plug-in, you can now read the passed dialog values from the *_data* variable.

Passing Data

Plug-ins contains Apply, Modify, Get, Enable/Disable methods for passing data from the dialog to the plug-in. These methods imitate functionality seen in Tekla Structures buttons.

- Apply function stores/sets new default values but doesn't modify the selected parts
- Modify modifies the selected parts as was defined in the plug-ins run method.
- The values are saved as with Apply if the dialog is closed with the OK button.

AttributeName

AttributeName is the name by which you can identify the control in the plug-in. If you set the control's AttributeName to "length", you can access it in the plug-in side with the corresponding attribute field in the *StructuresData* class.

```
[Tekla.Structures.Plugins.StructuresField("length")]
public double length;
```

AttributeNames in you Form needs to be different. Otherwise you will get “Failed loading plugin“ error message saying “An item with the same key has already been added.” Maximum length is 19 chars.

AttributeTypeName

AttributeTypeName is the name of the Tekla.Structures.Datatype type for the control's contents. In the example it is set as Distance. This setting affects the type of the variable (integer, double or string) that you get to the plug-in, as well as automatic formatting and localization of the field.

Plug-ins will always receive the data as Tekla Structures standard units, like millimeters for Distance. The following datatypes and attribute type data mappings are used:

DataType	**Plug-in attribute field type**
Boolean	integer
Distance	double
DistanceList	string
Double	double
Integer	integer
String	String (maximum 80 chars)

The Distance type for example formats the user input like in Tekla Structures and adds the corresponding unit abbreviation according to the current settings. If, for example, feet-inches units are active and the user types 14 in the field, the field will automatically change to say 1'-2". All the datatypes are defined in the Tekla.Structures.Datatype namespace in Tekla.Structures.Datatype.dll.

BindPropertyName

The *BindPropertyName* sets which property of the control the attribute binds to, hence which property the attribute is associated with.

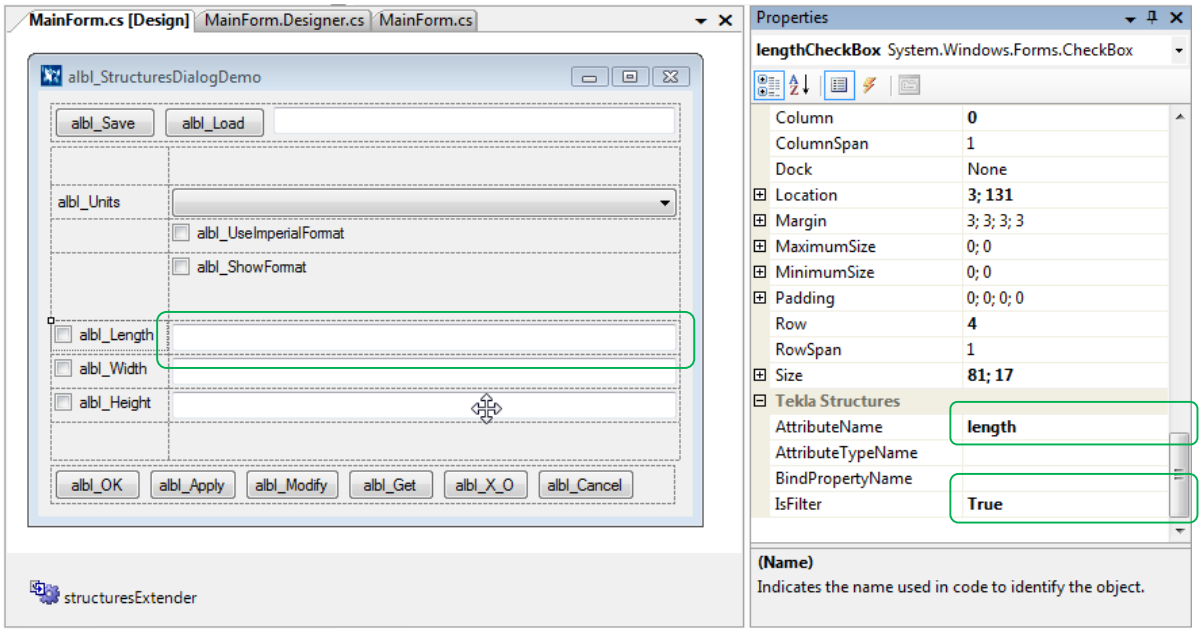
Most controls have default properties the attribute is bound to, e.g. *TextBox* controls have by default its *Text* property bound to its attribute and *ComboBox* controls are have by default its *SelectedIndex* property bound to its attribute.

Different attribute types (*AttributeTypeName*) limits which control properties an attribute are able to bind to.

IsFilter

According to the Tekla Structures dialog practice you should create an enable/disable checkbox for each control that the user can use to give input to your plug-in. If the field's checkbox is unchecked, the value for that attribute will be left unchanged when the user modifies the objects created with the plug-in.

The way to bind the checkbox to the corresponding control is to set the *AttributeName* property to the **same** value as the input control's *AttributeName*. Additionally you must set the *IsFilter* property to **true**. The AttributeTypeName should be left empty (see green markings in the picture below).



Control IsFilter property.

Default values

Default values for the fields in the dialog can be set as with classic Tekla Structures dialogs: create a standard-file and place it in the environment folder. Note that Tekla Open API dialogs use an XML format for their save files instead of the old native format.

Open your dialog from Tekla Structures, set the desired default values for the fields, set the file name to *standard* and hit Save. The file will be stored under the currently open model's folder, in the attributes subfolder. Its name is standard.<*AssemblyName*>.<*FormName*>.xml. Copy the file from there to the Tekla Structures environment structure.

Limitation

If a control property, bound to an attribute, is used directly from code it is strongly recommended to get or set the property value using *GetAttributeValue* and *SetAttributeValue*. Updating the property directly does NOT automatically update the bound attribute which might then be obsolete.

```
double example = GetAttributeValue<double>(ControlWithAttribute);
example += 5000.0;
SetAttributeValue(ControlWithAttribute, example);
```

Dialog design with INP

The user interface, or dialog, of a plug-in can be defined using the same definition language as custom components and system components: input file format (INP). See \TeklaStructures\<version>\applications\steel1\ts_page_1.inp for examples; the file xeng_page_1.inp in the same folder contains some documentation.

You can edit INP files without re-starting Tekla Structures by adding “set XS_DYNAMIC_INPUT_FILE=TRUE” to user.ini. An Update button appears on all INP based dialogs and pressing it will reload the INP definitions.

All plug-in INP files are re-created at startup. If you like you can disable this new behavior by adding the following line to teklastructures.ini:

- `set XS_DO_NOT_OVERWRITE_PLUGIN_INP_FILE=TRUE`

Naming the plug-in

The plug-in name will appear in the Component Catalog, and it has to be unique. You cannot have two plug-ins with the same name.

When using INP for dialog definition, plug-in name is defined following way:

```
[Plugin("BeamPlugin")]
    ...
    public class UserInterfaceDefinitions
    {
        public const string Plugin1 = @"" +
            page(""TeklaStructures"","""")
            {
                plugin(1, BeamPlugin)
                {
                    ...
                }
```

Note that in the above the plug-in names in bold must be the same. If a plug-in name has a space, i.e. "Beam Plugin", it is not possible to open dialog ("Beam Plugin.inp") from Tekla Structures Component Catalog.

Adding a thumbnail image

Usually thumbnail images in the component catalog show you a typical situation where the component can be used.

To add a thumbnail for the plug-in create an image, name it et_element_<pluginname> and save it in bmp format to ..\Tekla Structures\<version>\nt\bitmaps folder.

Special notices and reminders

- Make sure that the .dll created is in your nt\bin\plugins
- The plug-in's constructor must be defined as public.
- The references used in the plug-in have to point to the ones included under the correct Tekla Structures version: Tekla.Structures.Plugins, Tekla.Structures.Model, Tekla.Structures, etc.
- Plug-ins most likely won't work in any other Tekla Structures than the one they were created for; so, for using a plug-in in a different version it has to be compiled for that version
- Log file displays the number of found plug-ins.
- A plug-in should never call CommitChanges, since this would make undo very difficult for the user to do.

A step-by-step example

The following steps create a basic plug-in that uses Windows Forms.

Create a new Visual C# project; select "Class Library", and enter an appropriate name.

Add the following to References to the project:

```
nt\bin\dialogs\Tekla.Structures.Dialog.dll
nt\bin\plugins\Tekla.Structures.dll
nt\bin\plugins\Tekla.Structures.Datatype.dll
nt\bin\plugins\Tekla.Structures.Model.dll
nt\bin\plugins\Tekla.Structures.Plugins.dll
```

Rename Class1.cs to MainPlugin.cs (this could be a more specific name).

Edit MainPlugin.cs:

Add assemblies needed:

```
using TSDatatype = Tekla.Structures.Datatype;
using TSModel = Tekla.Structures.Model;
using TSPlugins = Tekla.Structures.Plugins;
```

Add:

```
public class StructuresData
{
}
```

Add:

```
[TSPlugins.Plugin("FormPlugin")]
[TSPlugins.PluginUserInterface("FormPlugin.MainForm")]
```

Inherit from PluginBase:

```
public class MainPlugin : TSPlugins.PluginBase
```

Add to MainPlugin class:

```
private readonly TSModel.Model _model;

public TSModel.Model Model
{
    get { return _model; }
}

private readonly StructuresData _data;

public MainPlugin(StructuresData data)
{
    // Link to model.
    _model = new TSModel.Model(true);

    // Link to input values.
    _data = data;
}

public override List<InputDefinition> DefineInput()
{
    // Define input objects.
}

public override bool Run(List<InputDefinition> input)
{
    try
    {
        // Write your code here.
    }
    catch (Exception e)
    {
        MessageBox.Show(e.ToString());
    }

    return true;
}
```

In MainPlugin.cs, add the following to the StructuresData class for each attribute on the dialog:

```
[TSPlugins.StructuresField("ATTRIBUTE_NAME")]
public VARIABLE_TYPE VARIABLE_NAME;
```

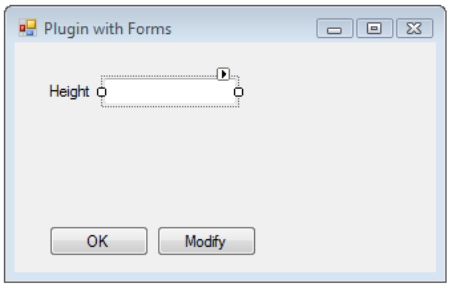
Replace ATTRIBUTE_NAME with the name entered for the AttributeName property, VARIABLE_TYPE with a type corresponding to AttributeNameType, and VARIABLE_NAME with a suitable variable name.

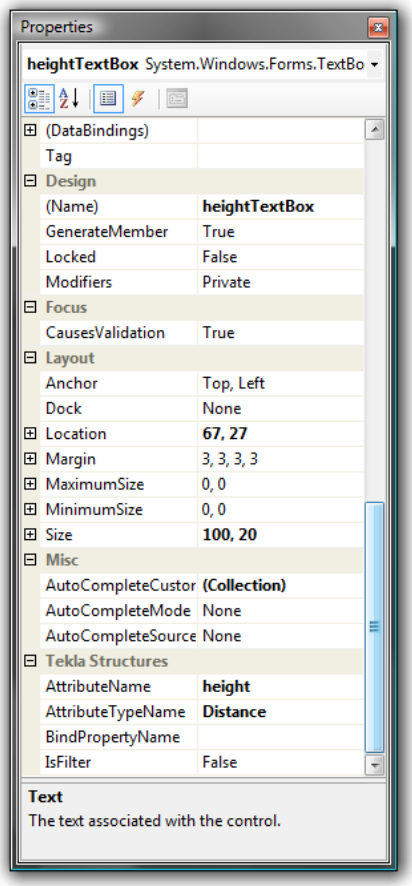
Add code to DefineInput() to prompt for input objects (parts, points, etc).

Add code to Run() to create parts.

Add a form (Project > Add Windows Form...), change the name to MainForm.cs or something specific.

Design the dialog. For each control on the dialog, enter a unique name for the AttributeName listed on the Properties panel (right-click on the control, select Properties); also, select an appropriate type for the AttributeNameType.





Add event handlers for the OK and Modify buttons. For OK, this.Apply() and this.Close() should be called; and for Modify, this.Modify() should be called.

```
private void okButton_Click(object sender, EventArgs e)
{
    this.Apply();
    this.Close();
}

private void modifyButton_Click(object sender, EventArgs e)
{
    this.Modify();
}
```

Build the project and copy the dll created to nt\bin\plugins.

Run Tekla Structures and look for the plug-in in the Component Catalog.

Drawing plug-ins

Drawing plug-ins are created by extending the *DrawingPluginBase* class for defining the plug-ins' business logic, and the *PluginFormBase* class for defining their dialog. The defining and retrieving inputs works a bit differently than in the model.

After the basic steps for creating a plug-in, the plug-in should have a .cs file for the plug-in code and a Form for the user interface. The drawing plug-in requires few mandatory fields in order to work.

Mandatory fields in the drawing plug-in

- StructuresData: gets the data from the user interface into the plug-in
- Constructor of the plug-in: copies the data from Tekla Structures
- DefineInput (public method): method that defines the input of the plug-in
- Run (public method): main method of the plug-in
- Plug-in definition attribute above the plug-in class

Dialog design with Windows Forms

Plug-in is defined by an attribute which is written prior to the plug-in class. The attribute sets also a name for the plug-in, e.g.

```
[Plugin("MarkPlugin")]
[PluginUserInterface("MarkPlugin.MarkPluginForm")]
```

User interface definition is not mandatory but if user interface is needed it is done with Windows Forms.

StructuresData

StructuresData works as a gateway to the interface. The attribute names of the controls in the interface should match the name in the plug-in code.

Supported data types are string, double and integer. Be sure to type the attribute name right because they are strings.

Constructor

Every plug-in needs a constructor. The constructor sets the StructuresData from the interface to be in use of the plug-in.

```
public SlotPlugin(SlotPluginData data)
{
        Data = data;
}
```

DefineInput()

DefineInput is a public method and it is one of the two methods which make the plug-in a plug-in. This method defines the input(s) of the plug-in. Drawing plug-ins supports more than one inputs. Plugins with zero inputs works also. The inputs are passed to the Run method as a List of InputDefinitions.
There is an InputDefinitionFactory that creates and retrieves input objects. They can be points or certain drawing objects (bolts and parts are supported). The method is executed when the drawing plug-in icon is pressed from Tekla Structures.

Inputs can be e.g. user based picked points or objects found by enumerating objects from views.

Run(List<InputDefinition> inputs)

Run is a public method and it is the main method of the plug-in. This is where the plug-in uses the inputs to affect Tekla Structures. StructuresData provides the information from the dialog of the plug-in and the list of inputs given as an argument provides the inputs from Tekla Structures.

Adding a drawing plug-in to Tekla Structures

Plug-in objects are added to drawings via toolbar icons. To add a drawing plug-in to a toolbar, follow these steps:

Make sure the compiled plug-in is in the \Program Files\Tekla Structures\<version>\nt\bin\plugins folder. The plug-in can be inside a folder.

While in the modeling-side, open the toolbar customization dialog (Tools -> Customize).

Locate and select your plug-in in the list on the left: there should be an item titled "Create xx", where xx is the name of the plug-in as specified with the "Plugin" tag in the class code.

Select any toolbar that is visible on the drawing-side from the list on the right or make a new toolbar for and set it visible, and click on the right-arrow between the lists. If the plug-in icon was added in a new toolbar, be sure to check that it is visible also in the drawing side toolbars.

To change the default icon shown in the toolbar create an image, name it according the name of your plug-in and save it in bmp format to ..\Tekla Structures\<version>\nt\bitmaps folder.

Close the toolbar customization dialog and open a drawing. An icon for adding a new plug-in object to the drawing should appear on the toolbar on which it was added.

Dialog opens when you double click the icon in the toolbar or when double clicking the plug-in component created to the drawing.

For more information and to start testing, please see the “MarkPlugin” code example in Start-up package that can be found on Extranet.

A step-by-step example

The following steps create a basic drawing plug-in that uses Windows Forms.

Create a new Visual C# project, select "Class Library", and enter an appropriate name.

Add the following to References to the project:

```
nt\bin\dialogs\Tekla.Structures.Dialog.dll
nt\bin\plugins\Tekla.Structures.dll
nt\bin\plugins\Tekla.Structures.Drawing.dll
nt\bin\plugins\Tekla.Structures.Plugins.dll
nt\bin\plugins\Tekla.Structures.Datatype.dll
```

Rename Class1.cs to MainPlugin.cs (this could be a more specific name).

Edit MainPlugin.cs:

Add assemblies needed:

```
using Tekla.Structures.Datatype;
using Tekla.Structures.Drawing;
using Tekla.Structures.Plugins;
```

Add StructuresData class

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

Add:

```
[Plugin("FormPlugin")]
[PluginUserInterface("FormPlugin.MainForm")]
```

Inherit from DrawingPluginBase:

```
public class MainPlugin : DrawingPluginBase
```

Add to MainPlugin class:

```
private StructuresData _data { get; set; }

public MainPlugin(StructuresData data)
{
    // Link to input values.
    _data = data;
}

public override List<InputDefinition> DefineInput()
{
    // Define input objects.
}

public override bool Run(List<InputDefinition> input)
{
    try
    {
        // Write your code here.
    }
    catch (Exception e)
    {
        // Handle exception.
    }

    return true;
}
```

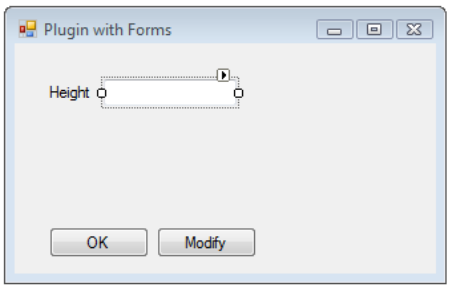
DefineInput() has to return a list of InputDefinitions created with InputDefinitionFactory.

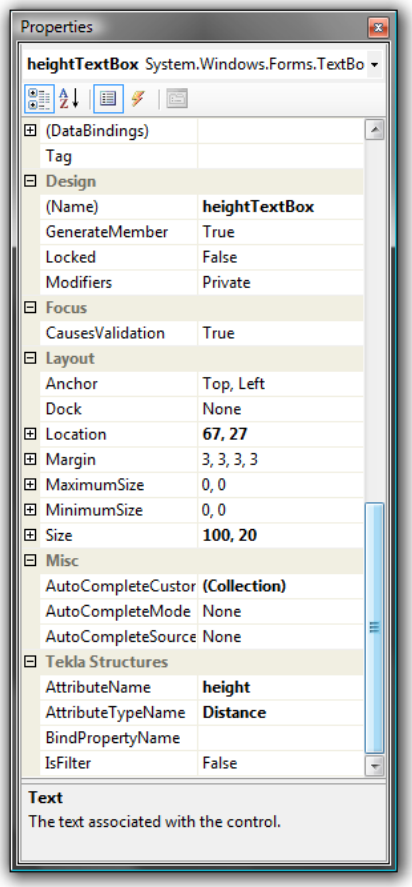
Add code to DefineInput() to prompt for input objects.

Add code to Run() to use the input points and objects.

Add a Form (Project > Add Windows Form...), change the name to MainForm.cs or something specific.

Design the dialog. For each control on the dialog, enter a unique name for the AttributeName listed on the Properties panel (right-click on the control, select Properties); also, select an appropriate type for the AttributeNameType. Remember to check the spelling in both ends.





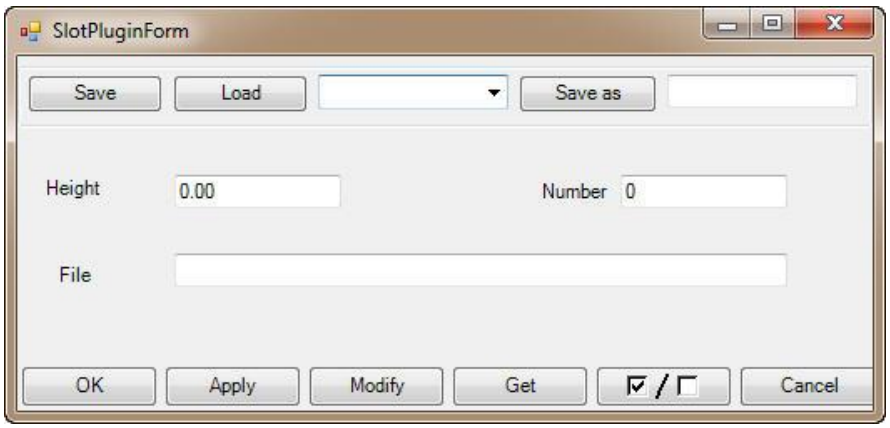
Add event handlers for the OK and Modify buttons. For OK, this.Apply() and this.Close() should be called; and for Modify, this.Modify() should be called.

```
        private void okButton_Click(object sender, EventArgs e)
        {
            this.Apply();
            this.Close();
        }

        private void modifyButton_Click(object sender, EventArgs e)
        {
            this.Modify();
        }
```

Build the project and copy the dll created to nt\bin\plugins folder.

Run Tekla Structures, open a drawing and press the shortcut to start the plug-in.



In order to get the “Save-Load-Save As” and “Ok-Apply-Modify-..” –buttons Tekla.Structures.Dialog.dll controls have to be added to Visual Studio Toolbox. Look from the chapter “User Interface Controls” how this is done.

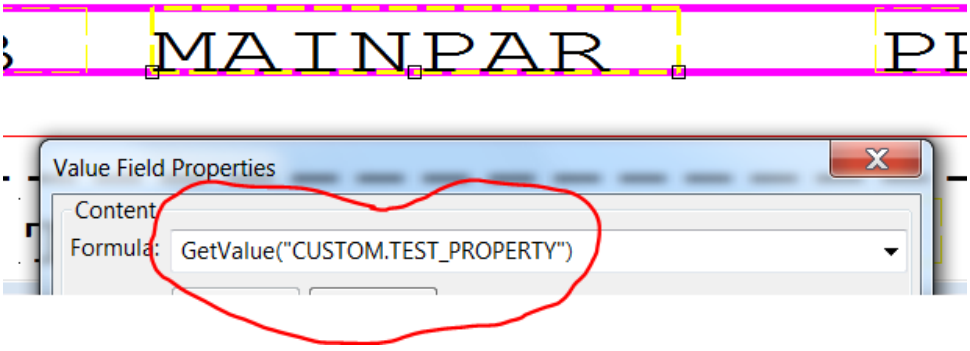
Example drawing plug-in MarkPlugin and SlotPlugin can be found from Start-Up packages Example solution.

Custom properties

Introduction

Custom property plug-ins are programmed tools for calculating template values used in reporting and drawings.

Custom properties are loaded inside the Tekla Structures process. Those are called using syntax “CUSTOM.xxx” in the same way as normal template properties are called.



The Tekla.Structures.CustomPropertyPlugin assembly provides `ICustomPropertyPlugin` interface which must be implemented when a custom property is defined:

```
public interface ICustomPropertyPlugin
{

    int GetIntegerProperty(int objectId);

    double GetDoubleProperty(int objectId);

    string GetStringProperty(int objectId);

}
```

In addition to interface implementation it is mandatory to define two custom attributes in implementation:

```
[Export(typeof(ICustomPropertyPlugin))]
[ExportMetaData("CustomProperty", "CUSTOM.NAME_OF_THE_PROPERTY")]
```

Example implementation of a custom property looks like this:

```
using System;
using System.ComponentModel.Composition;
using Tekla.Structures.CustomPropertyPlugin;

namespace CustomPropertyTest
{
    /// <summary>The test plugin for retuning modified id value.</summary>
    [Export(typeof(ICustomPropertyPlugin))]
    [ExportMetadata("CustomProperty", "CUSTOM.TEST_PROPERTY")]
    public class CustomPropertyTest : ICustomPropertyPlugin
    {
        /// <summary>Returns int value for object.</summary>
        /// <param name="objectId">The object id.</param>
        /// <returns>The <see cref="int"/>.</returns>
        public int GetIntegerProperty(int objectId)
        {
            return -1 * objectId;
        }

        /// <summary>Returns string value for object.</summary>
        /// <param name="objectId">The object id.</param>
        /// <returns>The <see cref="string"/>.</returns>
        public string GetStringProperty(int objectId)
        {
            return "Hello " + objectId.ToString();
        }

        /// <summary>Returns double value for object.</summary>
        /// <param name="objectId">The object id.</param>
        /// <returns>The <see cref="double"/>.</returns>
        public double GetDoubleProperty(int objectId)
        {
            return (double)(-1 * objectId);
        }
    }
}
```

Custom property functionality is using Managed Extensibility Framework (MEF) for loading custom property assemblies, therefore a reference to `System.ComponentModel.Composition` assembly is needed in the project.

It is possible to define several custom properties in one .NET assembly. However currently it is not possible to include custom properties and plug-ins in the same assembly.

Basic steps for creating a Custom property

- Create a new project in Visual Studio: Visual C#, Windows, Class Library.
- Microsoft .NET Framework 4.0
- Add references to `System.ComponentModel.Composition` and `Tekla.Structures.CustomPropertyPlugin` in Visual Studio. If other Open API assemblies are needed add needed references as well.
- Define needed metadata for custom property and implement the `ICustomPropertyPlugin` interface
- Methods in the interface are passing object ID as argument. Identifier of the object can be constructed using the ID in order to utilize the functionality in `Tekla.Structures.Model` assembly.
- The calculated value need to be returned back from the method.
- As non-mandatory requirement it is possible to add the type and formatting to setting file for Template Editor (such as contenattributes_global.lst). Please note some functionality in Tekla Structures (such as filtering) uses string property value as default unless specific type is defined in the setting file.
- The dll can be executed from the same location as model plug-ins

Special notices and reminders

- Make sure that the .dll created is in your nt\bin\plugins or in environments\common\extensions.
- The custom property class must be defined as public.
- The references used in the plug-in have to point to the ones included under the correct Tekla Structures version: Tekla.Structures.Plugins, Tekla.Structures.Model, Tekla.Structures, etc.
- Custom property plug-ins won't necessarily work in any other Tekla Structures than the one they were created for; so, for using a custom property in a different version may require compilation for that version
- Log file displays if loading of custom property is successful.
- Asynchronous actions are forbidden from custom properties. Value calculation must happen synchronously!
- Debugging of custom property happens in the same way as with plug-ins by attaching to Tekla Structures process and adding the break-point to the method.
- On the fly changes are not supported. A new dll requires restart of Tekla Structures.
- Please make sure no message boxes or pop-up dialogs are shown during the execution!!!

Notes for using Tekla Open API

TransformationPlane class

Change the work plane using TransformationPlane class.

```
Model myModel = new Model();
WorkPlaneHandler myWorkPlaneHandler = myModel.GetWorkPlaneHandler();
Beam myBeam = new Beam();
//Using the current work plane
TransformationPlane currentPlane =
      myWorkPlaneHandler.GetCurrentTransformationPlane();
myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(currentPlane);

//Using the object's local coordinate system
TransformationPlane beamPlane = new
      TransformationPlane(myBeam.GetCoordinateSystem());
myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(beamPlane);

//Using the global coordinate system
TransformationPlane globalPlane = new TransformationPlane();
myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(globalPlane);
```

MatrixFactory class

Transform a point between coordinate systems using MatrixFactory class.

```
Model myModel = new Model();
WorkPlaneHandler myWorkPlaneHandler = myModel.GetWorkPlaneHandler();
Beam myBeam = new Beam();
Point myPoint = new Point(100,200,300);

//Using the current work plane
TransformationPlane currentPlane =
      myWorkPlaneHandler.GetCurrentTransformationPlane();
myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(currentPlane);

//Get matrix to transform to the beam's local coordinate system
```

```
Matrix transformationMatrix =
      MatrixFactory.ToCoordinateSystem(myBeam.GetCoordinateSystem());

//Transform the point from current work plane to the local coordinate system

Point localPoint = TransformationMatrix.Transform(myPoint);


//Get the matrix to transform from local to current work plane

Matrix TransformationMatrix =
      MatrixFactory.FromCoordinateSystem(myBeam.GetCoordinateSystem());

//Transform the point from local to the current work plane coordinate system

myPoint = TransformationMatrix.Transform(localPoint);
```

Make sure what the current work plane is before using CoordinateSystem, Point and Offset classes.

CoordinateSystem class

When creating a new CoordinateSystem instance, the Origin, AxisX and AxisY are defined based on the current work plane coordinate system.

Point class

The beam's StartPoint and EndPoint properties return values in the current work plane coordinate system.

Offset class

Offset.Dx is defined using the object's coordinate system. Offset.Dy and Offset.Dz are defined using the current work plane coordinate system. Before trying to use the offset values it is important to make sure what the current work plane is since these values can be changed depends on the current work plane.

Picker class

Currently Picker.PickFace returns the face vertices in the global coordinate system. Picker.PickLine returns the points in the current workplane coordinate system.

Solid class

Solid.Intersect(LineSegment) : the line segment must be as least 3.2mm long.

Changing the work plane example

Following code shows how to change the current work plane to the top plane of the selected part. `CreateCoordinatesWithBeams()` helps to see what is the current work plane in the model.

```
private void button1_Click(object sender, EventArgs e)
{
        TSM.Model MyModel = new TSM.Model();
        TSM.UI.Picker MyPicker = new TSM.UI.Picker();
        TSM.Part MyPart =
        MyPicker.PickObject(TSM.UI.Picker.PickObjectEnum.PICK_ONE_PART)
                as TSM.Part;

        if (MyPart == null) return;

        // Save the current coordinate system
        // to get back to it when you are done.
        TSM.TransformationPlane SavedPlane =
                MyModel.GetWorkPlaneHandler().GetCurrentTransformationPlane();

        // Move into the selected part's coordinate system.
        MyModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(
                new TSM.TransformationPlane(MyPart.GetCoordinateSystem()));
        MyPart.Select();

        // The "Top" face of a part is in its positive Y direction,
        // so getting the solid's Max.Y is an easy way to find out
        // how far you need to move your origin to get to that face.
        TSM.Solid MySolid = MyPart.GetSolid();
        double OffsetY = MySolid.MaximumPoint.Y;
        TSG.Point MyOrigin = new TSG.Point(0.0, OffsetY, 0.0);

        // You'll need X & Y vectors for the next step.
        TSG.Vector Xvector = new TSG.Vector(1, 0, 0);
        TSG.Vector Yvector = new TSG.Vector(0, 1, 0);

        // Set the new coordinate system.
        MyModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(
                new TSM.TransformationPlane(MyOrigin, Xvector, Yvector));

        // Visual representation of the current coordinate system
        // in the model.
        CreateCoordinatesWithBeams();

        // Do whatever else you need to do while in the new coordinate system.

        // Reset the original coordinate system.
        MyModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(SavedPlane
);
}

public static void CreateCoordinatesWithBeams()
{
        Point point = new Point(0, 0, 0);
        Point point2 = new Point(1000, 0, 0);
        Beam beamX = new Beam();
        beamX.StartPoint = point;
        beamX.EndPoint = point2;
        beamX.Class = "2";
        beamX.Profile.ProfileString = "D10";
        beamX.Position.Depth = Position.DepthEnum.MIDDLE;
        beamX.Position.Plane = Position.PlaneEnum.MIDDLE;
        beamX.Insert();

        point = new Point(0, 0, 0);
        point2 = new Point(0, 1000, 0);
        Beam beamY = new Beam();
        beamY.StartPoint = point;
        beamY.EndPoint = point2;
        beamY.Class = "3";
        beamY.Profile.ProfileString = "D20";
        beamY.Position.Depth = Position.DepthEnum.MIDDLE;
        beamY.Position.Plane = Position.PlaneEnum.MIDDLE;
        beamY.Insert();
```

```
        point = new Point(0, 0, 0);
        point2 = new Point(0, 0, 1000);
        Beam beamZ = new Beam();
        beamZ.StartPoint = point;
        beamZ.EndPoint = point2;
        beamZ.Class = "4";
        beamZ.Profile.ProfileString = "D30";
        beamZ.Position.Depth = Position.DepthEnum.MIDDLE;
        beamZ.Position.Plane = Position.PlaneEnum.MIDDLE;
        beamZ.Insert();
}
```

Handling exceptions in applications/plugins

Follow the steps below to create safe applications/plugins.

- Minimize to use IO in the methods called when a application/plug-in is loaded by Tekla Structure. Make sure it cleans up all IO and memory so there is no problem that would cause Tekla Structure to hang.
- Keep the constructor as simple as possible.
- Check any input from the user, and if it is not what is required, either ask again or exit.
- For errors where you need to terminate the application or plug-in, throw the exception or let the highest level method catch it.
- Only exit the application from the highest level method. In case a plug-in, it should be the plug-ins Run method and make sure the Run method handles all exceptions.

User Interface Controls

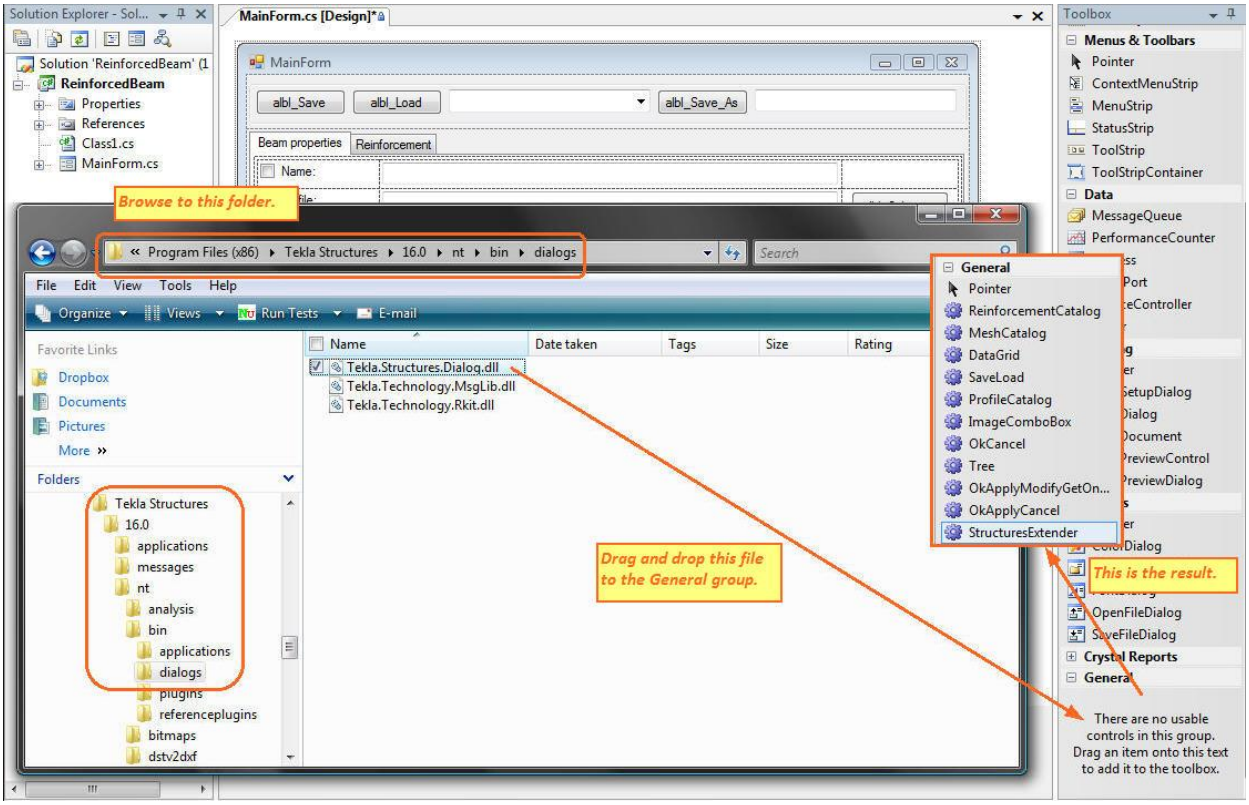
Tekla Open API contains Windows Forms dialogs for several catalogs to help developers of .NET applications and plug-ins which have Forms and need to provide for example profile selection in UI.

Also “Save-Load-Save as” buttons and “OK-Apply-Modify-Get” buttons are available as custom controls.

Both buttons and catalog custom controls can be found from Tekla.Structures.Dialog.UIControls namespace. The namespace contains also dialog templates.

You can drag and drop a custom control from Visual Studio’s Toolbox into a Form. There are two ways to add controls to Visual Studio’s Toolbox:

- Right click on the Toolbox and select “Choose items”. Then click the browse button to open Tekla.Structures.Dialog.dll. The controls will be added to the list. You can select one control or all the controls to be added to the Toolbox.
- Use Windows Explorer to navigate to the \Program Files\Tekla Structures\<version>\nt\bin\dialogs\Tekla.Structures.Dialog.dll, open the Designer and drag and drop the dll into the General tab of the Toolbox. All the controls will be added to the Toolbox. See the snapshot below.



Catalog custom controls

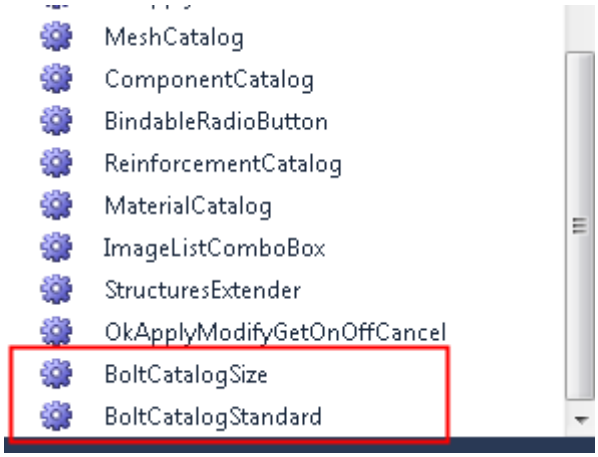
When a catalog button is clicked, the control will do the enumeration of the catalog and show the items in a dialog. You can also create a new instance of the profile / reinforcement / mesh selection dialog, just by passing a list of the items you want to show in the dialog.

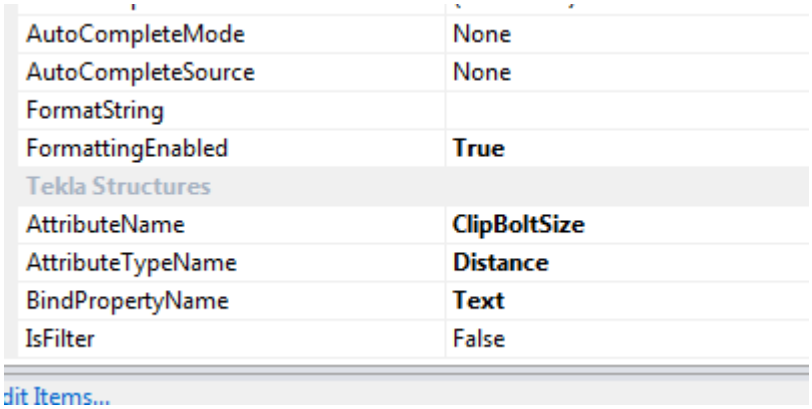
The item that user selects will be stored to a public property (SelectedProfile, SelectedMesh, SelectedMeshName, SelectedMeshGrade, SelectedRebarSize, SelectedRebarGrade, SelectedRebarBendingRadius).

Setting the public property to a certain value will select the value from the dialog when the dialog opens (if the value is included in the catalog).

For more information, see the “ReinforcedBeam” code example and Self Learning material, both can be found from the Start-up package on Extranet.

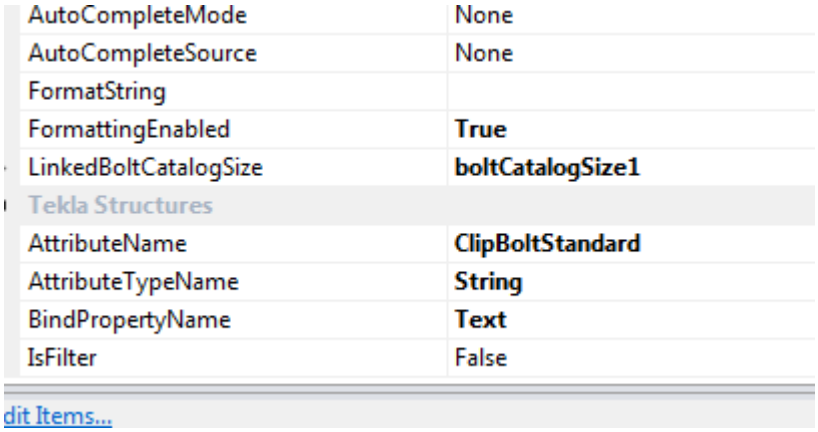
BoltCatalog





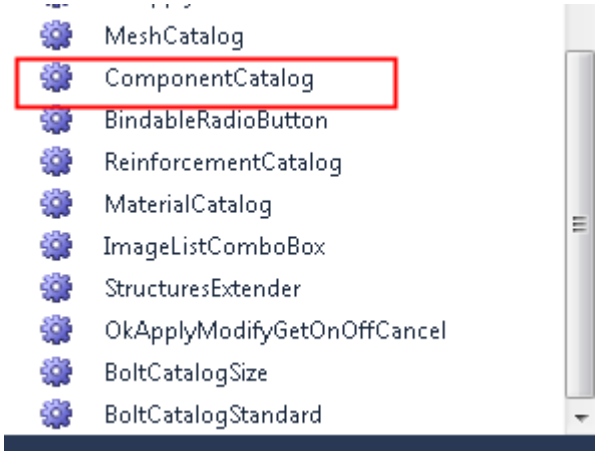
In properties panel of BoltCatalogSize

Select the BoltCatalogSize which will be paired with the BoltCatalogStandard in LinkedBoltCatalogSize property.

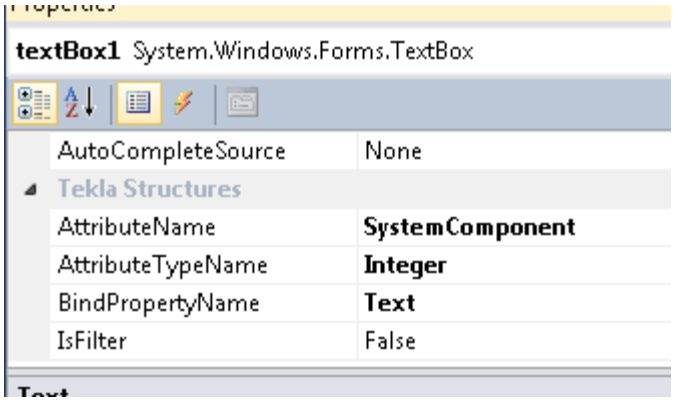


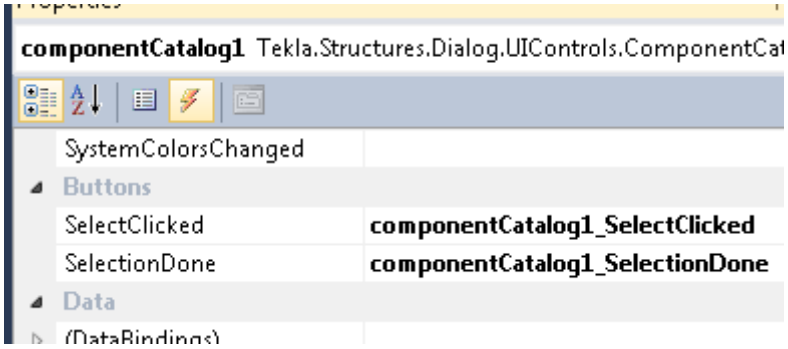
In properties panel of BoltCatalogStandard

ComponentCatalog



For the system components identified by the number

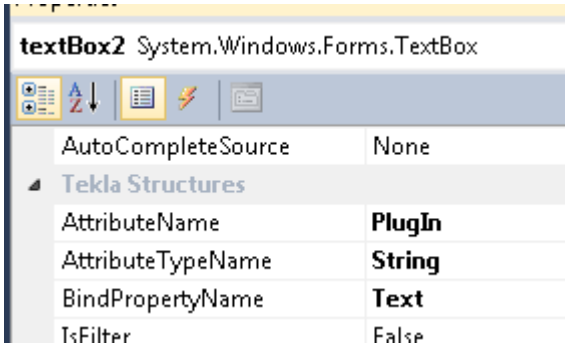


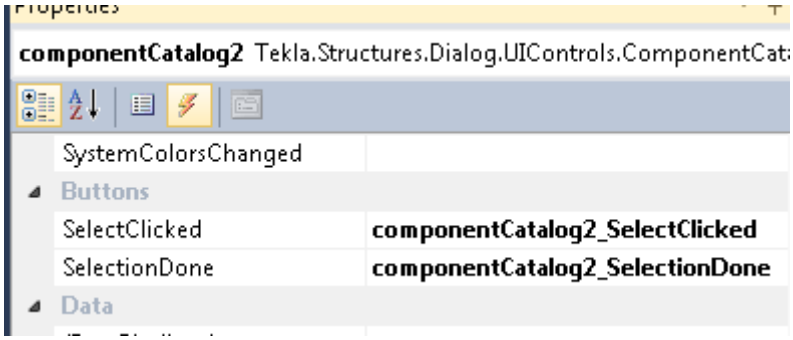


```
private void componentCatalog1_SelectClicked(object sender, EventArgs e)
{
    componentCatalog1.SelectedNumber =
        string.IsNullOrEmpty(textBox1.Text) ? Constants.XS_DEFAULT : int.Parse(textBox1.Text);
}

private void componentCatalog1_SelectionDone(object sender, EventArgs e)
{
    SetAttributeValue(textBox1, componentCatalog1.SelectedNumber);
}
```

For the plugins / custom components indentified by the name






```
private void componentCatalog2_SelectClicked(object sender, EventArgs e)
{
    componentCatalog2.SelectedName = textBox2.Text;
}

private void componentCatalog2_SelectionDone(object sender, EventArgs e)
{
    SetAttributeValue(textBox2, componentCatalog2.SelectedName);
}
```

Localization of strings

Localization means here translating and enabling the product for a specific language.

Localization of plug-in name in Component Catalog

It is possible to get the name of plug-ins translated in the Component Catalog.

The unique plug-in name is defined in the code in this statement:

```
[Plugin("RebarPluginSample")] //RebarSample3 example in StartUpPackage
```

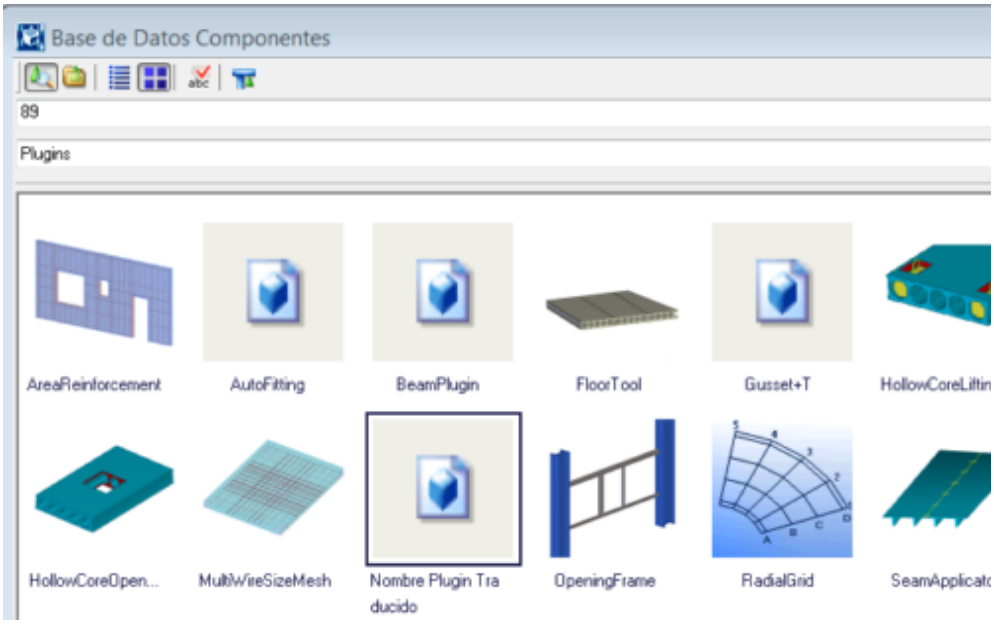
The translated strings for Tekla Structures components are stored in the file plugins.ail. This file is located in Tekla Structures installation folder, for example C:\Program Files\Tekla Structures\<version>\messages.

Continuing with the previous example, these lines are added at the bottom of plugins.ail:

```
String_utf8 RebarPluginSample
{

    entry = ("esp", "Nombre Plugin Traducido");
    entry = ("enu", "Translated Plugin Name");
};
```

The plug-in name in the component catalog is now translated.



Localization of plug-in or application strings

The localization of strings is done through XML-files, used only by your application or plug-in.

Editing

The localization file must have XML content (like shown below) and must be of UTF-8 format.

```
<?xml version="1.0" standalone="yes"?>
<Table>

<MsgLibStrings>
    <StrName>albl_MyString1</StrName>
        <chs>My String 1***</chs>
        <cht>My String 1***</cht>
        <csy>My String 1***</csy>
        <deu>My String 1***</deu>
        <esp>My String 1***</esp>
        <fra>My String 1***</fra>
        <hun>My String 1***</hun>
        <ita>My String 1***</ita>
        <jpn>My String 1***</jpn>
        <nld>My String 1***</nld>
        <plk>My String 1***</plk>
        <ptb>My String 1***</ptb>
        <ptg>My String 1***</ptg>
        <rus>My String 1***</rus>
        <enu>My String 1</enu>
</MsgLibStrings>

<MsgLibStrings>
    <StrName>albl_MyString2</StrName>
        <chs>My String 2***</chs>
        <cht>My String 2***</cht>
        <csy>My String 2***</csy>
        <deu>My String 2***</deu>
        <esp>My String 2***</esp>
```

```
        <fra>My String 2***</fra>
        <hun>My String 2***</hun>
        <ita>My String 2***</ita>
        <jpn>My String 2***</jpn>
        <nld>My String 2***</nld>
        <plk>My String 2***</plk>
        <ptb>My String 2***</ptb>
        <ptg>My String 2***</ptg>
        <rus>My String 2***</rus>
        <enu>My String 2</enu>
</MsgLibStrings>

</Table>
```

It is recommended to use Visual Studio when creating or making changes (adding, changing or removing strings) to a localization file. Make sure that the XML file is *well formed* and *valid*.

Files open in Visual Studio is automatically checked to be *well formed*, meaning the tags are symmetrical.

Multiple lines

When you have a long string that is split into multiple lines make sure that there are no extra new lines or spaces that are easily added by some XML editors: extra new lines and spaces are shown also in the user interface.

The string should NOT look like this:

```
<StrName>albl_Create_drawings</StrName>
<enu>
            Select parts in the model.
            Select a master drawing in the list below.
            Then click this button to create drawings.
        </enu>
```

The string should be like this:

```
<StrName>albl_Create_drawings</StrName>
<enu>Select parts in the model.
Select a master drawing in the list below.
Then click this button to create drawings.</enu>
```

Microsoft Word and ellipsis

An ellipsis is a row of three periods (…).

Do not use Microsoft Word for adding or editing XML strings because AutoCorrect feature replaces ellipses automatically by a one character “â€ ¦”.

If you for example add a string “Loading drawings…” using Word, it will look like this:

```
<enu>Loading drawingsâ€¦</enu>
```

Step-by-step instructions

A localization file is automatically loaded for plug-ins (Forms that inherit from PluginFormBase) if the file name and path of the XML file is the same as for the plug-in.

To localize an application that inherits from ApplicationFormBase, or plug-in which XML file name and path doesn’t match to plug-in, you must

- set the language

- load the localization file(s)
- call Localization.Localize() for your Form

```
Dialogs.SetSettings(string.Empty);
Localization.Language =
         (string)Tekla.Structures.Datatype.Settings.GetValue("language");

Localization.LoadFile("CommonStrings.xml");
Localization.LoadFile("MyApplicationStrings.xml");

Localization.Localize(myForm);
```

Note that in case you need many XML files, you can read an arbitrary amount of them by calling Localization.LoadFile() method for each of them.

Message box localization

Message box message strings and titles are localized with the localization property's GetText method like this:

```
MessageBox.Show(Localization.GetText("MyMessage"),
    Localization.GetText("MyTitle"));
```

Strings in message box buttons (like OK and Cancel) cannot be localized. Translations of those buttons come automatically according to the selected language of the operating system.