Tekla Open API

Examples

Tekla Structures 2016

Contents

Preface

Here are small examples of different topics.

If you are inexperienced with Tekla Open API, then you should first go through the exercises for beginners.

(..\TeklaOpenAPIStartUpPackage_v201\SelfLearning\SelfLearning.pdf)

The examples are divided into three parts: applications, plug-ins and scripts/macros.

Application is a program that has separate executable, which is started outside Tekla Structures.

Plug-in is a system component (DLL) that can be executed from component catalog. It runs inside Tekla Structures.

Macros/Scripts are generally actions that you can record while working with Tekla Structures. Macros are basically C# (.cs) source files that are compiled at run-time, which could be edited; applications and plug-ins are compiled executables or DLLs.

The example projects are located in the sub-folders inside the Examples folder, and are also organized in a Microsoft Visual Studio solution: Examples.sln.

NOTE: More examples have been added after this document was created; so, they are not described in detail here. However, each project contains a ReadMe.txt file that briefly explains the purpose of the example; they are found in the sub-folders and in the Examples.sln.

1 Application examples

1.1 Beam creation example

Location

All necessary source files for compiling this application are found in the folder "Examples\Model\Applications\BeamApplication" in StartupPackage.

Purpose

This example shows in detail how to create a very simple .NET application which utilizes the Model API to create a beam in a Tekla Structures model from a button in application window.

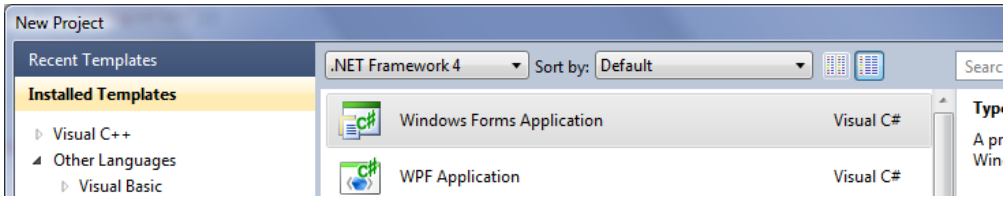
Setup of Tekla Structures .NET project

This document aims to describe how to setup a new .NET project in Microsoft Visual Studio using the Tekla Open API.

In the following example we will show how to create a small application from scratch that has a single button that inserts a Beam into a Tekla Structures model.

1. Create a new Windows Forms Application project.

The first step to take is to start Microsoft Visual Studio and creating a new project. (**New -> Project** from the **File** menu)

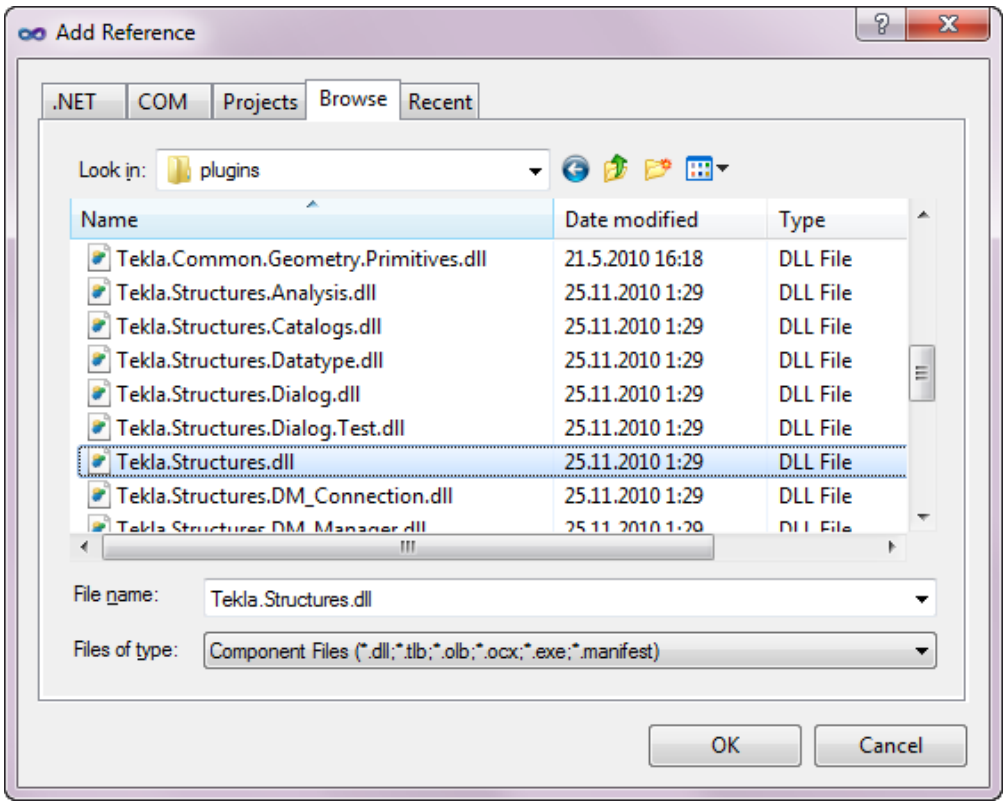


Fill out the details as shown in the screenshot and press **OK** to create an empty Windows application.

2. Add references to Tekla Open API assemblies.

Once you have created the project, you need to add a reference to the Tekla Open API assemblies you will use, like **Tekla.Structures.Model.dll** and **Tekla.Structures.dll**. Adding these references will enable you to use objects included in those assemblies.

To add the references you can either right-click on **References** in the Solution Explorer and select **Add Reference** from the pop-up menu, or you can select **Add Reference…** from the **Project** menu.



Click **Browse…** in the **Add Reference** menu to locate the Tekla.Structures.Model.dll and Tekla.Structures.dll files. You will find these files in the **\nt\bin\plugins** folder of your Tekla Structures version. Once you have located these select them both and select **Open** and press **OK** in the **Add Reference** dialog.

From now on you are able to use the classes and methods of Tekla Open API in your project.

3. Add directives to namespaces of Tekla Structures assemblies

View the code of Form1.cs (right-click on the form and select **View code**) and add the *directive* lines:

```
using Tekla.Structures.Model;
using TS = Tekla.Structures.Geometry3d;
```

at the top of the page.

4. Add button to application form and beam creation to Click-event

Go back to the design view of the form and add a button to it. Place the button on the form and name it **Create Beam**. Double-click the button to open up the code for its Click event.

```
private void button1_Click(object sender, EventArgs e)
{
    Model myModel = new Model();

    Beam myBeam = new Beam(new TS.Point(1000, 1000, 1000),
                           new TS.Point(6000, 6000, 1000));
    myBeam.Material.MaterialString = "S235JR";
    myBeam.Profile.ProfileString = "HEA400";
    myBeam.Insert();
    myModel.CommitChanges();
}
```

The following code can now be added to the click method. It will insert a Beam between the points (1000,1000,1000) and (6000,6000,1000) in the model.

Explanation

A brief explanation of the code above and what it does:

1. Creates a new Model object that represents the Tekla Structures Model we have opened in Tekla Structures.
2. Checks if we have a Tekla Structures Model that we can connect to.
3. Creates the two points that will be used as the start and end point for the beam.
4. Creates a new Beam based on the two input points.
5. Sets the Beams Material and Profile.
6. Inserts the Beam into the Tekla Structures Model.
7. CommitChanges makes sure all changes that have been done are updated in Tekla Structures and that the model view is updated accordingly. If this command isn't run then you will have to refresh the view manually to see the changes.

After this has been done all you need to do is make sure you have a Tekla Structures Model opened and after choosing **Start** from the **Debug** menu in Visual Studio to run your project. Your form with the **Create Beam** button should now appear and pressing the **Create Beam** button should place a Beam at the wanted coordinates into the model.

Testing the Beam creation application in example

You can test the application in Tekla Structures as follows:

1. Open a Tekla Structures model.
2. Start the application.
3. Click the button on application window.

The result of the application shall be something as shown here.


B
A
1
2
Z

1.2 Reinforcement example #1

This example creates single bars and stirrups.

Location

Together with this document you should have the source codes and the project files for Visual Studio in “Examples\Model\Applications\RebarExamples”.

Content of folders and files:

- Macro – Tekla Structures macros
- RebarSample1 – source files and project files for example #1
- RebarSample2 – source files and project files for example #2
- RebarSample3 – source files and project files for example #3
- RebarExamples.sln – solution file for Visual Studio

Explanation

In this first example we are using some of the basic reinforcement classes. The actual application has been implemented as an executable (file RebarSample1.exe) but practically the same source code can be run as Tekla Structures macro (file RebarSample1B.cs). One advantage of a macro is that it can be run both within Tekla Structures 18.0 and 21. versions without any modifications. The executables shall be compiled separately for both versions; however, the source code is still 100% compatible between 18.0 & 21..

In the next paragraphs we are going through the most important aspects of the program code.

Setting up the connection between the application and Tekla Structures

Before you can access any of the Tekla Open API objects, you need to initialize the connection between Tekla Structures and your application. This is done by creating a new instance of class Tekla.Structures.Model:

```
Model myModel = new Model();
```

Enumerating Tekla Structures model objects

In many practical applications the most fundamental user input is the user 's pre selection; i.e., user makes some selection model view and runs the application which is doing something to or with those selected objects. In this example we will enumerate all selected objects of Beam type.

```
ModelObjectEnumerator myEnum =
myModel.GetModelObjectSelector().GetSelectedObjects();

while(myEnum.MoveNext())
{
    Beam myPart = myEnum.Current as Beam;
```

```
    if(myPart != null)
    {
            .
            .
            .
    }
}
```

Management of current work plane

In most practical cases it makes sense to set the current work plane so that the geometrical calculations are straight forward. In this example we will set the work plane parallel to the beam's local coordinate system. This way we can ensure that the behavior and output of the application is not dependent on the beam's location or orientation in 3D model.

Whenever the application is changing the current work plane it is a good practice that the previous work plane set by user will be restored.

```
// first store current work plane
TransformationPlane currentPlane =
myModel.GetWorkPlaneHandler().GetCurrentTransformationPlane();

// set new work plane same as part's local coordsys
TransformationPlane localPlane = new
TransformationPlane(myPart.GetCoordinateSystem());

myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(localPlane);
.
.
.
// remember to restore current work plane
myModel.GetWorkPlaneHandler().SetCurrentTransformationPlane(currentPlane);
```

Getting the solid geometry data

In this example we are not concentrating on solid geometry and how it can be used in application development. For this reason, only the minimal solid data i.e. solid boundary data will be used. However, to get the boundary corner coordinates we need a reference to the part's solid geometry data.

```
// get solid of part to be used for rebar point calculations
Solid solid = myPart.GetSolid() as Solid;
```

Creating longitudinal reinforcement bars

In this example, we create the reinforcement bars by using the class SingleRebar. This class represents single reinforcement bar in model and UI of Tekla Structures.

First, we create the new instance of the class and set the basic property values. In this example we are using hard coded values but in real applications the values will be retrieved from some calculations or user input.

```
// initialize the single rebar object to be used in longitudinal bar creation
SingleRebar bar = new SingleRebar();
bar.Father = myPart;
bar.Size = "20";
```

```
bar.Grade = "A500HW";
bar.OnPlaneOffsets.Add(0.0);  // please note the data type has to be 'double'
bar.FromPlaneOffset = 0.0;
bar.Name = "Longitudinal";
bar.Class = 7;
bar.EndPointOffsetType =
Reinforcement.RebarOffsetTypeEnum.OFFSET_TYPE_COVER_THICKNESS;
bar.EndPointOffsetValue = 25.0;
bar.StartPointOffsetType =
Reinforcement.RebarOffsetTypeEnum.OFFSET_TYPE_COVER_THICKNESS;
bar.StartPointOffsetValue = 25.0;
```

After we have initialized the object properties we can calculate the input points for rebar shape polygon. To keep the example very simple we are using the solid boundary points.

```
// create longitudinal bars at four boundary corners of the solid
// bar #1 at "lower left"
bar.Polygon.Points.Add(new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y +
40, solid.MinimumPoint.Z + 40));
bar.Polygon.Points.Add(new Point(solid.MaximumPoint.X, solid.MinimumPoint.Y +
40, solid.MinimumPoint.Z + 40));
```

Once we have the object fully defined, we can insert the object into model as follows. Now we have created a new reinforcement bar into Tekla Structures model.

```
bar.Insert();
```

After we have created the first rebar, we are re-using the same object to create three other rebars. To do this, we just clear the shape polygon and add new points to it. When we have added the necessary point, we will call the insert method.

```
// bar #2 at "lower right"
bar.Polygon.Points.Clear();
bar.Polygon.Points.Add(new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y +
40, solid.MaximumPoint.Z - 40));
bar.Polygon.Points.Add(new Point(solid.MaximumPoint.X, solid.MinimumPoint.Y +
40, solid.MaximumPoint.Z - 40));
bar.Insert();

// bar #3 at "upper right"
bar.Polygon.Points.Clear();
bar.Polygon.Points.Add(new Point(solid.MinimumPoint.X, solid.MaximumPoint.Y -
40, solid.MaximumPoint.Z - 40));
bar.Polygon.Points.Add(new Point(solid.MaximumPoint.X, solid.MaximumPoint.Y -
40, solid.MaximumPoint.Z - 40));
bar.Insert();

// bar #4 at "upper left"
bar.Polygon.Points.Clear();
bar.Polygon.Points.Add(new Point(solid.MinimumPoint.X, solid.MaximumPoint.Y -
40, solid.MinimumPoint.Z + 40));
bar.Polygon.Points.Add(new Point(solid.MaximumPoint.X, solid.MaximumPoint.Y -
40, solid.MinimumPoint.Z + 40));
bar.Insert();
```

Creating stirrup reinforcement

In principle, the creation of stirrups follows the same logic as the longitudinal bars. However, in this specific case we are using the class RebarGroup as it is the most effective way to model stirrups.

```
First, create the new instance of the class and initialize the basic
properties.

// initialize the rebar group object for stirrup creation
RebarGroup stirrup = new RebarGroup();
stirrup.Father = myPart;
stirrup.Size = "8";
stirrup.RadiusValues.Add(16.0);
stirrup.Grade = "A500HW";
stirrup.OnPlaneOffsets.Add(20.0);  // please note the data type has to be
'double'
stirrup.FromPlaneOffset = 50;
stirrup.Name = "Stirrup";
stirrup.Class = 4;
stirrup.EndPointOffsetType =
Reinforcement.RebarOffsetTypeEnum.OFFSET_TYPE_COVER_THICKNESS;
stirrup.EndPointOffsetValue = 20.0;
stirrup.StartPointOffsetType =
Reinforcement.RebarOffsetTypeEnum.OFFSET_TYPE_COVER_THICKNESS;
stirrup.StartPointOffsetValue = 20.0;
stirrup.StartHook.Angle = 135;
stirrup.StartHook.Length  = 80;
stirrup.StartHook.Radius = 16;
stirrup.StartHook.Shape = RebarHookData.RebarHookShapeEnum.HOOK_90_DEGREES;
stirrup.EndHook.Angle = 135;
stirrup.EndHook.Length = 80;
stirrup.EndHook.Radius = 16;
stirrup.EndHook.Shape = RebarHookData.RebarHookShapeEnum.HOOK_90_DEGREES;

// set group spacing
stirrup.Spacings.Add(250.0);
stirrup.SpacingType =
Reinforcement.RebarSpacingTypeEnum.SPACING_TYPE_TARGET_SPACE;
```

The actual shape of the stirrup and the range of the group is defined by two polygons. In this example, we create two polygons representing stirrup shapes at both ends of the group. After the new polygon is instantiated we add the necessary points to them.

```
// set the polygon and insert stirrup into model
Polygon polygon1 = new Polygon();
polygon1.Points.Add(new Point(solid.MinimumPoint.X, solid.MaximumPoint.Y,
solid.MinimumPoint.Z));
polygon1.Points.Add(new Point(solid.MinimumPoint.X, solid.MaximumPoint.Y,
solid.MaximumPoint.Z));
polygon1.Points.Add(new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y,
solid.MaximumPoint.Z));
polygon1.Points.Add(new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y,
solid.MinimumPoint.Z));
polygon1.Points.Add(new Point(solid.MinimumPoint.X, solid.MaximumPoint.Y,
solid.MinimumPoint.Z));

Polygon polygon2 = new Polygon();
polygon2.Points.Add(new Point(solid.MaximumPoint.X, solid.MaximumPoint.Y,
solid.MinimumPoint.Z));
polygon2.Points.Add(new Point(solid.MaximumPoint.X, solid.MaximumPoint.Y,
solid.MaximumPoint.Z));
polygon2.Points.Add(new Point(solid.MaximumPoint.X, solid.MinimumPoint.Y,
solid.MaximumPoint.Z));
polygon2.Points.Add(new Point(solid.MaximumPoint.X, solid.MinimumPoint.Y,
solid.MinimumPoint.Z));
polygon2.Points.Add(new Point(solid.MaximumPoint.X, solid.MaximumPoint.Y,
solid.MinimumPoint.Z));
```

Finally, after the polygons are added to array RebarGroup.Polygons, the stirrup can be created by inserting the object into Tekla Structures model.

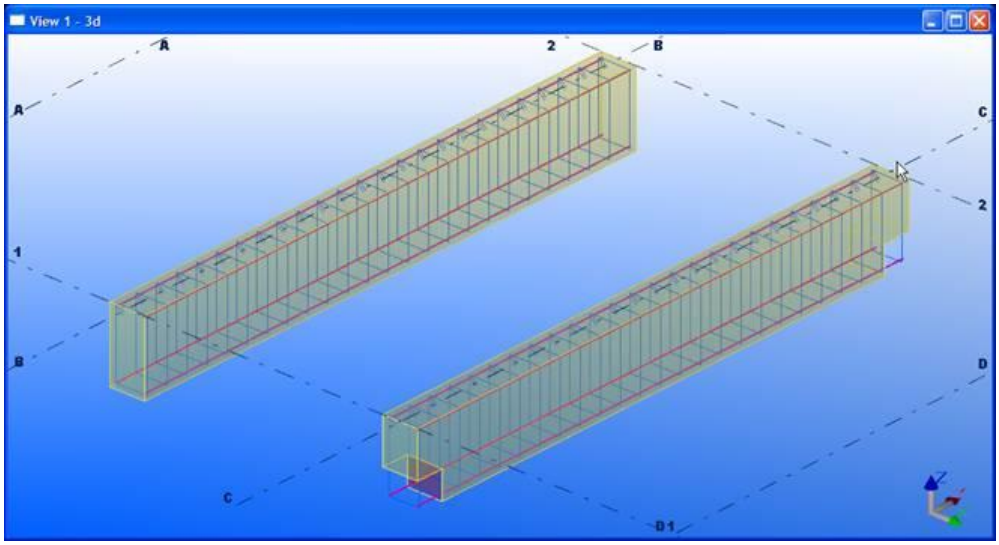
```
stirrup.Polygons.Add(polygon1);
stirrup.Polygons.Add(polygon2);
stirrup.Insert();
```

Testing the application in example #1

You can test the application in Tekla Structures as follows:

1. Open a Tekla Structures model.
2. Create some rectangular beams.
3. Select the beams in model view.
4. Run the application RebarSample1.exe simply by double clicking the file in Windows Explorer. Please make sure you double click the file located in the folder that matches your Tekla Structures version.
5. Alternatively you can run the macro 'RebarSample1B' by selecting the option **Tools->Macros**… , selecting the macro form the list, and clicking button 'Run' in the 'Macros' dialog. Please note that before you can see the macro in the list you need to install (=copy) the file 'RebarSample1B.cs' into your macro folder.

The result of the application shall be something as shown here.



Characteristics of example #1

This is a static application, meaning that it creates the bars and stirrup based on situation as it was during the moment the application was run. If you make changes to beam size or length, the rebars are not automatically adjusted. To be able to get the reinforcement updated you need to modify it interactively or you need to delete the reinforcement and re-run the application.

As you can see in the above picture the application is not able to manage the end notches in anyway. This is of course because of the very simple geometry handling. In the next example, we will be using one of the simple system components which would add more intelligence to our application.

1.3 Reinforcement example #2

This example creates single bars and strirrups using system connections.

Location

Together with this document you should have the source codes for sample programs and the project files for Visual Studio in "Examples\Model\Applications\RebarExamples".

Content of folders and files:

- RebarSample2 – source files and project files for example #2
- RebarExamples.sln – solution file for Visual Studio

Explanation

In this second example we are using two simple system components. With the help of these components we can actually increase the intelligence of the application due to their ability to manage notches and openings in the part.

The basic structure of the application is the same as in example one. The source code of the standalone application can be found in '*RebarSample2.cs*'. The corresponding macro can be found in '*RebarSample2B.cs*'.

In the next paragraphs we are going through parts of the program code which are different from example one.

Using system component to create longitudinal bars

First of all we create new instance of the class Component.

```
// initialize the component used to model longitudinal bars
Component component1 = new Component();
component1.Number = 30000070; // unique number for "longitudinal rebars"
                              // component
```

Then we will initialize the objects properties. For components we can utilize the "save as" files and load the base values for all properties and only set the necessary values explicitly.

```
// mamange the settings i.e. load standard defaults and set the
// few important attribute values explicitly
component1.LoadAttributesFromFile("standard");
component1.SetAttribute("bar1_no", 4);      // number of bars
component1.SetAttribute("cc_side", 45.0);   // cover thickness at side
component1.SetAttribute("cc_bottom", 45.0); // cover thickness at bottom
```

Next we set up the input for the component. The input sequence shall be same as the input when the component is applied interactively. In this case the input contains the part and three points. Point class is included to "Tekla.Structures.Geometry3d" namespace which is included with

```
// prepare input points for the "longitudinal rebars" component
Point p1 = new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y,
    solid.MaximumPoint.Z);
Point p2 = new Point(solid.MaximumPoint.X, solid.MinimumPoint.Y,
    solid.MaximumPoint.Z);
Point p3 = new Point(solid.MinimumPoint.X, solid.MinimumPoint.Y,
    solid.MinimumPoint.Z);

// set up component input sequence
ComponentInput input1 = new ComponentInput();
input1.AddInputObject(myPart);
input1.AddTwoInputPositions(p1, p2);
input1.AddOneInputPosition(p3);

// Add the input for component
component1.AddComponentInput(input1);
```

Finally we are ready to insert the component into Tekla Structures model.

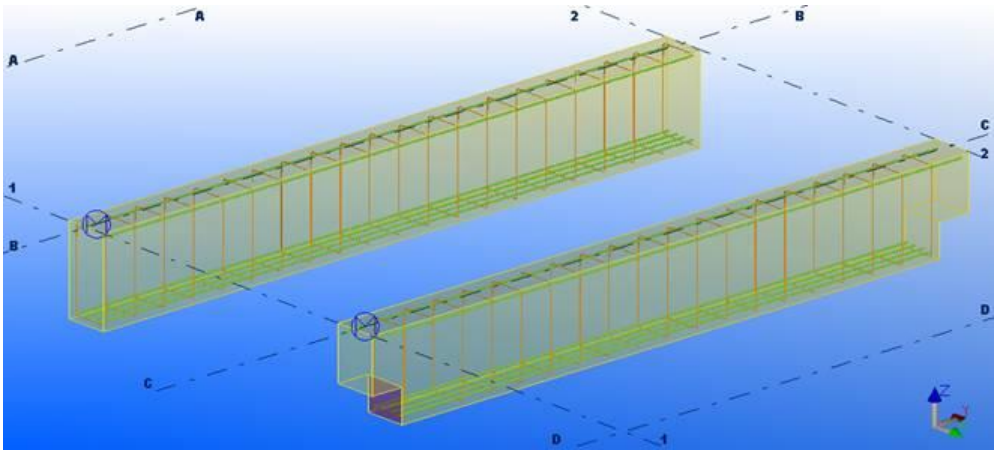
```
// insert new component instance into model
component1.Insert();
```

Testing the application in example #2

You can test the application in Tekla Structures as follows:

1. Open a Tekla Structures model.
2. Create some rectangular beams.
3. Select the beams in model view.
4. Run the application '*RebarSample2.exe*' simply by double clicking the file in Windows explorer view. Please make sure you double click the file locating in the folder matching with your Tekla Structures version.
5. Alternatively you can run the macro *'RebarSample2B'* by selecting the option **Tools→Macros**… , selecting the macro form the list and clicking button **Run** in the *Macros* dialog. Please note that before you can see the macro in the list you need to install (=copy) the file *'RebarSample2B.cs'* into your macro folder.

The result of the application shall be something as shown here.



Characteristics of the Example #2

As we can see in the picture, the application is able to recognize the notches at beam end which causes the longitudinal bar at the bottom being shorter like the stirrup range.

This is still a static application like the first one; however, unlike basic reinforcement objects the components adds some intelligence into our model. For example, adding or deleting the notches at beam ends will cause the components to be updated and the reinforcement adjusted.

1.4 SplitPolygonWeld

This example splits an existing polygon weld into its segments.

Location

All necessary source files for compiling application are found from folder "Examples\Model\Applications\SplitPolygonWeld" in the Open API StartupPackage.

Explanation

Split Polygon Weld is an example application showing how to first use Picker class to pick the weld and then how to handle the weld points in order to implement the splitting. This example utilizes also method Operation.CopyObject.

Application contains following functionality:

- **Split polygon weld:** Splits the polygon weld.

Testing the example application

You can test the application in Tekla Structures as follows

1. Open a Tekla Structures model.
2. Run the application by double clicking the SplitPolygonWeld.exe file in Windows Explorer.
3. Click the “Split polygon weld” and select polygon weld in model view.

1.5 SimpleDrawingList

This example finds a list of drawings.

Location

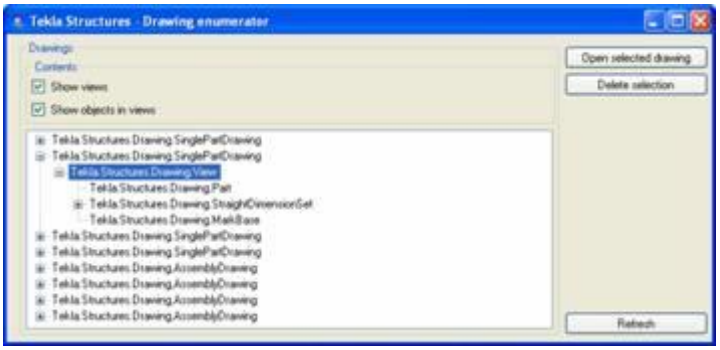
All necessary source files for compiling application are found from folder "Examples\Drawings\Applications\SimpleDrawingList" in StartupPackage.

Explanation

Simple Drawing List is an example application showing how to go through various drawings and drawing objects. This application is not intended to provide any meaningful functionality to Tekla Structures; it is only a simplified example of the Drawings API.

Application contains following functionality:

- **Refresh:** When you press this button is pressed, the tree view is re-generated. Depending on the amount of drawings this might take a while.
- **Show views:** When this check box is selected, the application will fetch the views in the drawings. This is considerably slower than just fetching the drawings.
- **Show objects in views:** When this check box is selected, the application will fetch the objects in the views. *Show view* must be selected to enable this.
- **Open selected drawing:** When you press this button, the selected drawing in the tree view is opened. If nothing is selected or if the selected tree view item is not a drawing, this button will do nothing.
- **Delete selection:** When you press this button, the selected item on the tree view is deleted from the drawing database. This button deletes both drawings and drawing objects.



Testing the example

You can test the application in Tekla Structures as follows:

1. Open a Tekla Structures model.
2. Run the application SimpleDrawingList.exe simply by double clicking the file in Windows explorer view. Please make sure you double click the file locating in the folder matching with your Tekla Structures.
3. Select enumeration mode and press "Refresh" to update list.

1.6 BasicViews

This example creates views in drawings.

Location

All necessary source files for compiling application are found from folder "Examples\Drawings\Applications\BasicViews" in StartupPackage.

Explanation

Create Basic Views is an example application showing how to create general arrangement drawings with different views. It uses the same definitions for the view rotations as the model command "Add component basic views". This should not be mixed with single part or assembly or cast unit drawing views, which have they own definitions for views. Drawings created with this application are not automatically updated. Therefore, if the basic views are created from assembly or from task, these drawings must be recreated after adding new parts to the assembly or task.

Application contains following functionality:

- **Create:** Creates drawings from the selected model objects. If there are several model objects selected, one drawing is created for each model object. Model object can be part, assembly, task, component or cast unit.
- **Open drawings:** If this check box is checked, drawing creation is visible. If there are multiple model objects selected, the last drawing is left open when the application has completed all drawings. When this check box is not selected, the drawings are created in invisible mode similar to normal Tekla Structures drawing creation.
- **Top/Front/End/3d view:** When selected, the specified view is created.

Testing the example application

You can test the application in Tekla Structures as follows

1. Open a Tekla Structures model.
2. Select parts in model view.
3. Run the application BasicViews.exe simply by double clicking the file in Windows explorer view. Please make sure you double click the file locating in the folder matching with your Tekla Structures.

2 Plug-in examples

2.1 Reinforcement example #3 – plug-in

This example is the same as Reinforcement example #2, but as a plug-in.

Location

Together with this document you should have the source codes for sample programs and the project files for Visual Studio in "Examples\Model\Applications\RebarExamples".

Content of folders and files:

- RebarSample2 – source files and project files for example #2
- RebarExamples.sln – solution file for Visual Studio

Explanation

In this third example, we are implementing a new plug-in component. We are using the same method to create the reinforcement as we have used in example #2, and by looking at the source code in file *'RebarSample3.cs'*, you can see that it uses the system components in the same way.

In the next paragraphs we are going through parts of the program code which are different from previous examples.

Declaration of class to manage plug-in component properties

First of all, we declare a new class which will contain the properties we need for our plug-in. These properties will be stored in Tekla Structures model with every instance of out plug-in component. By including the specific meta code statements, we can define the names for those properties as they appear in Tekla Structures model.

```
public class StructuresData
{
    [Tekla.Structures.Plugins.StructuresField("bottom_number")]
    public int BottomNumber;
    [Tekla.Structures.Plugins.StructuresField("top_number")]
    public int TopNumber;
}
```

In this example, we only have two properties which control the number of bars at top and bottom.

Plug-in name in component catalog

Every plug-in component will have a unique name in Tekla Structures component catalog. This name is defined with following meta code statement. Together with the plug-in component name we have to define the container variable for the user interface. This will be discussed more at paragraph 4.4

```
[Plugin("RebarPluginSample")]
[PluginUserInterface(RebarPluginSample.UserInterfaceDefinitions.dialog)]
```

Plug-in class declaration

To declare the actual plug-in class, we need to derive the class from virtual PluginBase class.

```
public class RebarPluginSample: PluginBase
{
    private StructuresData data;
    .
    .
```

As you can see, we have also declared one member variable data which will be explained later. This plug-in class needs to implement the virtual constructor which is taking its parameter an object of class StructuresData:

```
public RebarPluginSample(StructuresData data)
{
    this.data = data;
}
```

Definition of input

This plug-in component will define its input sequence by overriding the method DefineInput(). In this method, we will collect the input by using the Picker class and its various methods and by adding the input into an ArrayList, returned as result of the method.

```
// this is called by TS when the command is started
public override ArrayList DefineInput()
{
    Picker  picker = new Picker();
    ArrayList inputList = new ArrayList();

    ModelObject o1 = picker.PickObject(Picker.PickObjectEnum.PICK_ONE_PART);

    InputDefinition input1 = new InputDefinition(o1.Identifier);

    inputList.Add(input1);

    return inputList;
}
```

Introduction of user interface definition and user interface

Since this plug-in component has properties, we need to define the property dialog for it. This is done by defining with a meta code statement the static string variable which holds the user interface definition (inp code). The definition is based on same inp-format as the dialogs of custom components.

```
[PluginUserInterface(RebarPluginSample.UserInterfaceDefinitions.dialog)]

public class UserInterfaceDefinitions
// this is a nested class containing the UI definition for plug-in component
// the format for UI definition is same as for custom components
{
    public const string dialog =
        @"page(""TeklaStructures"","""")
        {
            plugin(1, ""RebarPluginSample"")
            {
                tab_page("""", ""Parameters 1"", 1)
                {
                    parameter(""Number of bottom bars"", ""bottom_number"",
                        integer, number, 1)
                    parameter(""Number of top bars"", ""top_number"",
                        integer, number, 4)
                }
                depend(2)
                modify(1)
                draw(1, 100.0, 100.0, 0.0, 0.0)
            }
        }";
}
```

Implementation of the Run() method

The heart of the plug-in is the overridden Run() method.

```
public override bool Run(ArrayList Input)  // this is called by TS when the
component is
                                           // created or modified
{
. . .


}
```

In this method we are using the Tekla.Net API to create the reinforcement in similar way as in example #2. However, there are certain differences because of the reason how plug-ins are working.

First of all, as you can see there is no enumeration of selection. The reason for this is that the part to be reinforced is given as input for the plug-in and we can get the reference to part by simply selecting it with the identifier we can get from the input.

```
Identifier id = InputDefinition)Input[0]).GetInput() as Identifier;
Part myPart = myModel.SelectModelObject((id) as Part;
```

Default value handling

As we wanted to add few properties for our plug-in we need to take care of the default value handling. In Tekla Structures it is possible to leave the input empty and the intention is that the plug-in or system component is expected to choose the value based on it's best knowledge.

In simple cases the plug-in can use some fixed value but in general it will be somehow dependent on input objects. For example in our case the number of bars may be dependent on beam size, however here we are using a simple method with fixed values.

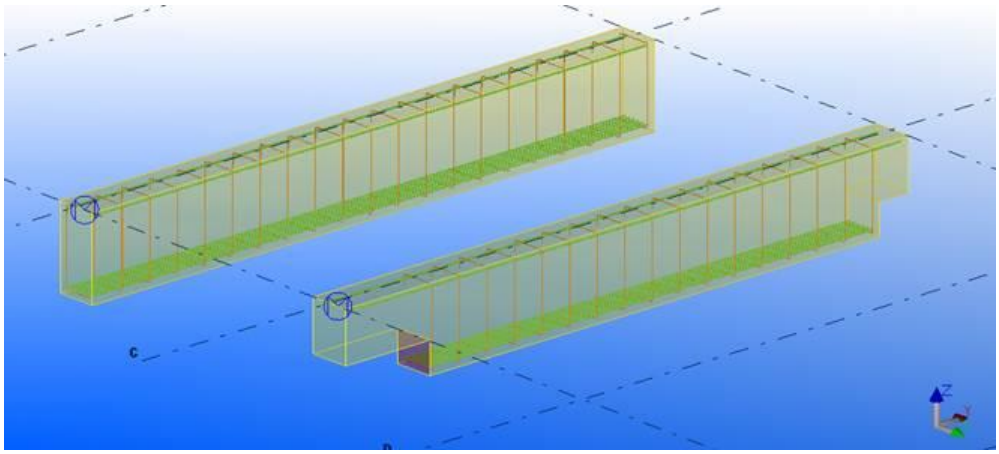
```
// we need to calculate final values for input properties which have default
(=empty)
// input values.
if(IsDefaultValue(data.BottomNumber))
    data.BottomNumber = 6;

if(IsDefaultValue(data.TopNumber))
  data.TopNumber = 2;
```

Testing the plug-in

You can test the new plug-in component in Tekla Structures as follows:

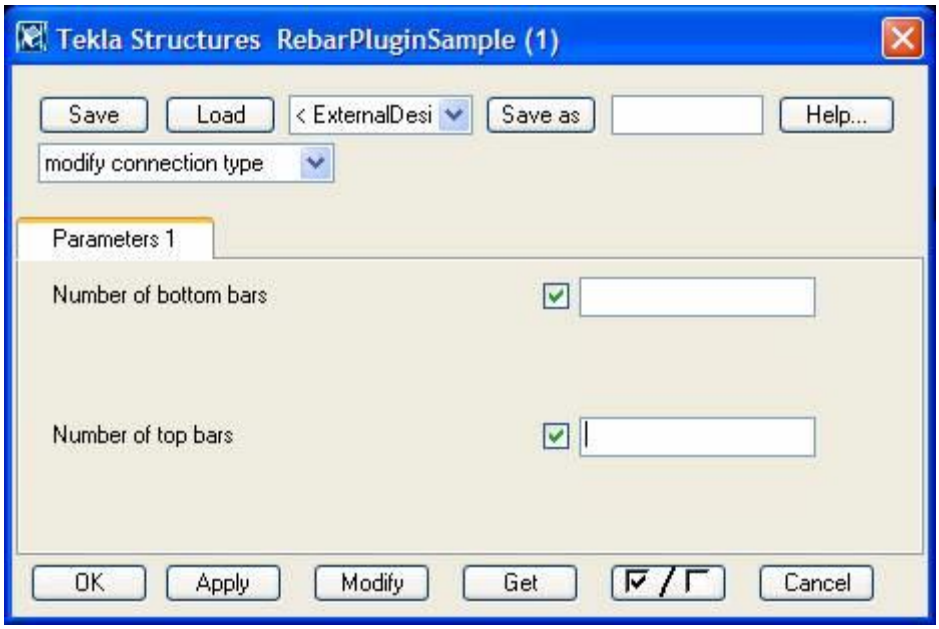
1. Check that your Tekla Structures has been setup to support Tekla.net based plug-in components. See Appendix A for details on how to do the setup manually.
2. First of all copy the file 'RebarSample3.dll' into **nt\bin\plugins\myplugins**-folder.
3. Start Tekla Structures
4. Open Tekla Structures model
5. Create some rectangular beams
6. Open component catalogue (Ctrl+F) and search component 'RebarPluginSample'
7. Click the component in the catalogue and pick beam in the model view

The result of the plug-in component shall be something as shown here.



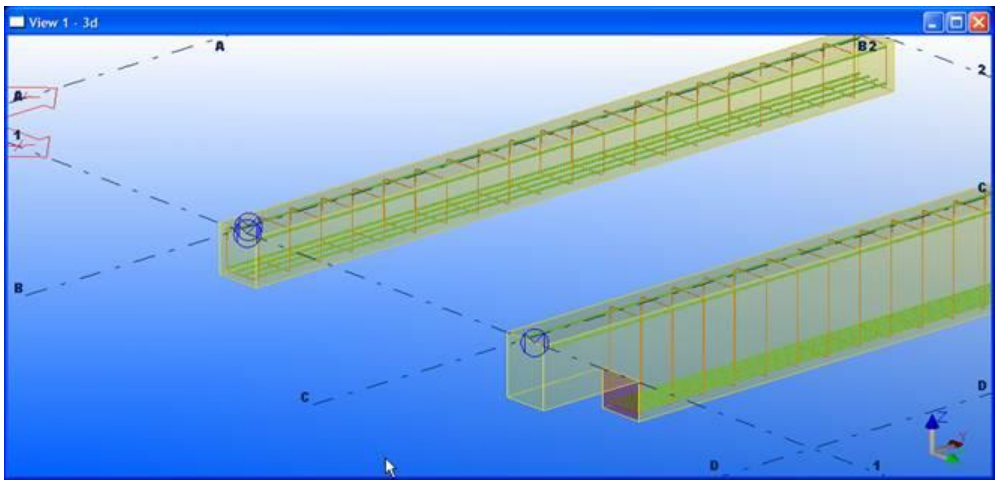
Characteristics of example #3

Compared to previous examples, this example has few advantages. First of all we have now few properties we can adjust. By double clicking the reinforcement in model view the following dialog will open.



Here, we can modify the properties of the plug-in component the same way as the properties of custom components or system components.

Secondly, whenever some geometrical changes happen to the beam, the plug-in is automatically called and the reinforcement will be adjusted.



2.2 SplicePlugin

This example creates beams and splice connections between them.

Location

All necessary source files for compiling application are found from folder "Examples\Model\Plugins\SplicePlugin" in StartupPackage.

Purpose

SplicePlugin is simple example that creates beams and then creates splice connections between them. This plug-in uses the Europe environment; this can be changed by using other profiles, bolt standards and bolt sizes.

Explanation

The code is divided into 2 parts. First, the beams are created, and then each splice is created. Creation of a splice can be divided again into two parts: creation of plates and then creation of bolts.

The Run()-function does these:

- Calculate amount of beams needed
- Create those beams in loop and add those to array
- Call CreateSplices() with the array as argument

The CreateSplices function loops through the beams and does these, starting from the second beam:

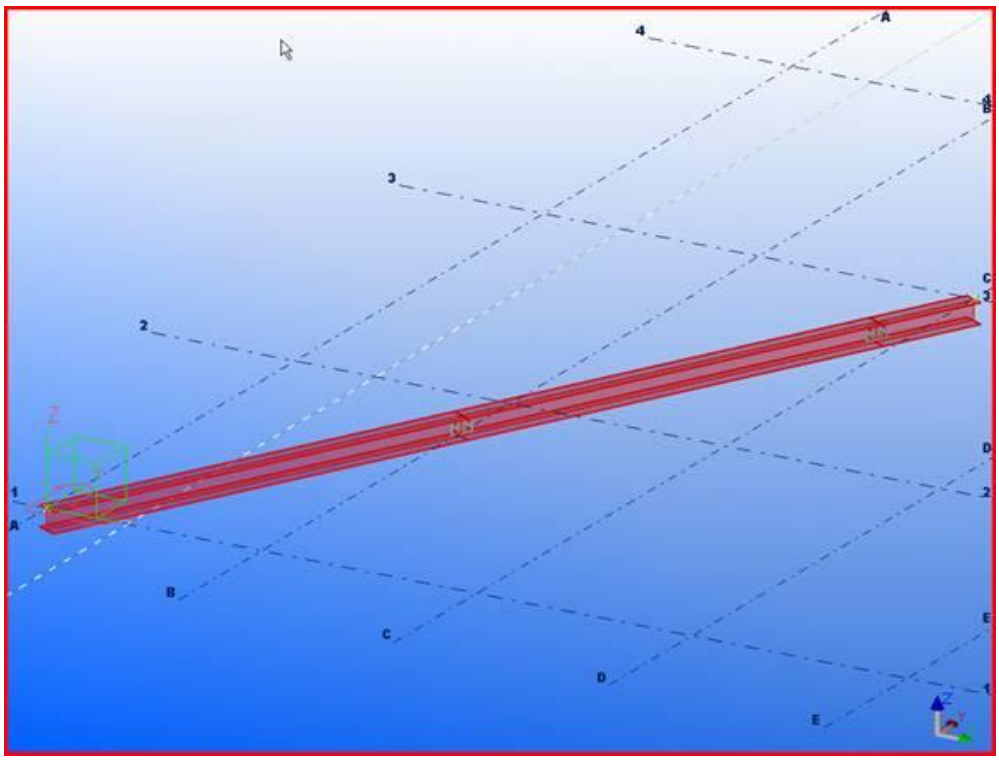
- Create two plates
- Create bolt array

Using the plug-in

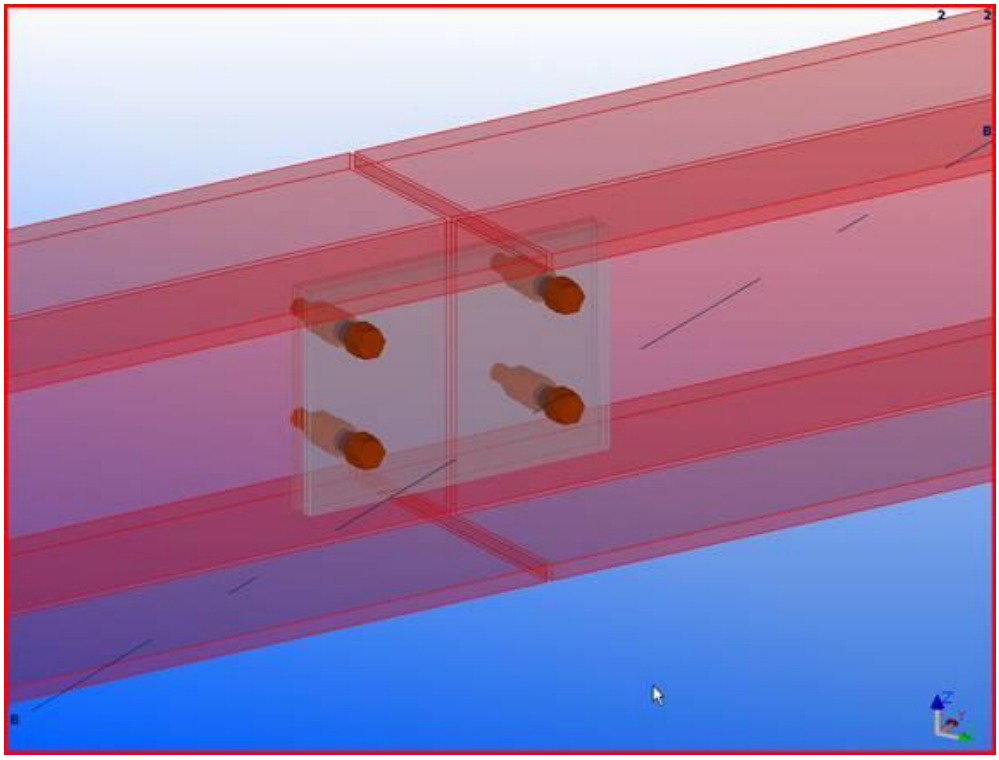
1. Select SplicePlugin from the component catalog.
2. Select two points in the model.

The plug-in creates 6000mm long beams between those points and between the beams, splices are created. The splices are constructed from two plates and four bolts, two bolt arrays.

The result should look like this:



A close-up of one of the splices:



Known problems

If the last beam is too short, the result might look strange; a plate could stand out from the beam.

3 Scripts/Macros

3.1 Dimensioning Center of Gravity

This code example, “Examples\Scripts\DimensionedCOG.cs”, loops through every object in every view of an opened drawing, and adds dimensions for the parts’ Center of Gravity (COG in the code). Actual, Center of Gravity is asked from Tekla Structures by using template properties.

3.2 RebarExample1B

This is the same as Rebar example 1, but as macro: Model\Applications\RebarExamples\Macro\RebarSample1B.cs.

3.3 RebarExample2B

This is the same as Rebar example 2, but as macro: Model\Applications\RebarExamples\Macro\RebarSample2B.cs.