Jacaranda Flame
Consulting

HANLON
INDUSTRIES

TECHNICAL DOCUMENTATION

Tekla Structures - 3D Optimisation Project (Phase IV)

Prepared by group 10:

TERRY WANG : 500495108
ARIELLA ZHAO : 500521119
JOSHUA LING : 510468017
Zeeshan Ansari : 510370813
Zhe Sun : 490051469

Aim

This document mainly explains and instructs users who have access to the source code material and desire to modify the codes to their likings. As this program have been extensively developed, it is crucial to understand the overall code infrastructure for any further developments beyond phase III.

The primary goal of our software is to facilitate and streamline the design of standard billboard structures. By utilizing standardized inputs, we aim to accelerate the design process and eliminate the need for modeling simple, repetitive structures manually. Developed as a Windows Forms Application, our program is powered by C# programming language and makes use of the TeklaOpenAPI library.

Getting Started

Installation - Setup for coding environment

1. Unpacking the project

Unzip the compressed file containing the project file and other code files and open it in Visual Studio by manually selecting the solution or project in Visual Studio (**If you have opened Visual Studio First**) or directly clicking the solution file shown in **Figure 1**.

Figure 1: Solution/Project File direct access in your local folder

2. Installing necessary packages (Tekla Structure)

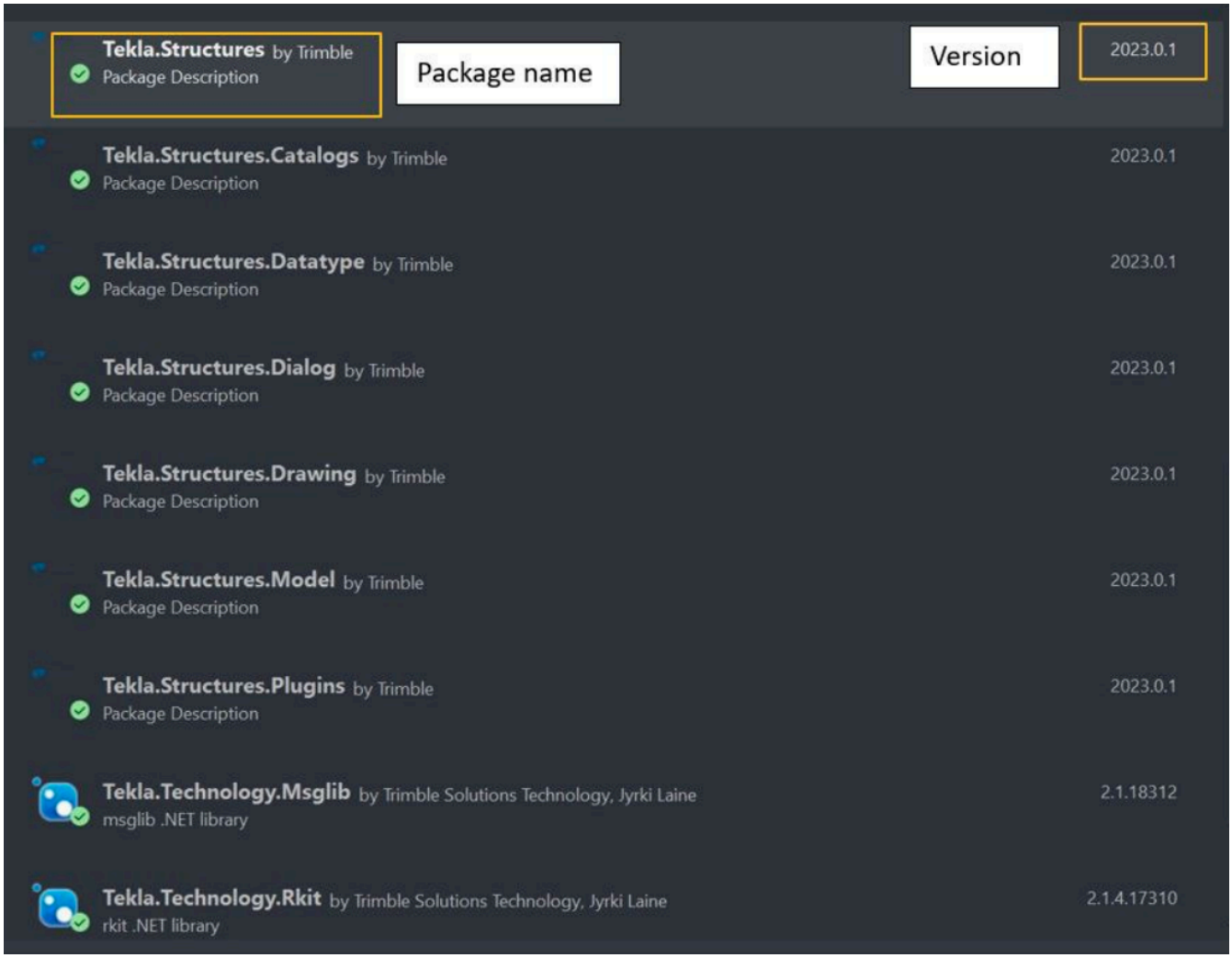
The program requires the TeklpaOpenApi (2021 Depcrecated version) and Tekla Structures Packages (version **should match** the version of Tekla Structures - the year number indicated on your installed Tekla Structures version. E.g. 2023 - Tekla Structures 2023) as well as some other packages that are recommended by visual studio.

To install Tekla OpenAPI + Tekla Structures packages in Visual Studio:

1. Menu Bar (Tools)
2. NuGet Package Manager -> Manage NuGet Packages for Solution… Search for ‘TeklaOpenAPI’ in the search bar. Make sure you are in the Browse tab. If your do not have the search result, make sure your Package source in nuget.org on the top right.
    a. If you do not have nuget.org Package source, open the settings (the cogwheel next it)
    b. Add a new Package source by clicking the + icon and entering the details as shown. Name: nuget.org
    c. Source: https://api.nuget.org/v3/index.json

3. Select the project to install
4. Click install for all packages in the current solution
5. If you run into any issues, check out this link: https://developer.tekla.com/teklastructures/documentation/get-started-tekla-structures-open-api

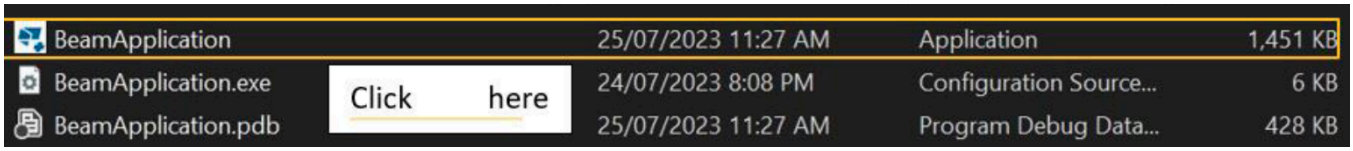
These Packages should be installed in your solution, please check the version of the packages.



Usage

Running the application

There are different ways of running the program. The zipped files will include an executable which has already been compiled and built. You can simply run the executable and run, if you have already opened Tekla Structures, otherwise an error message will display.



Compile and run

This method is compulsory if any modification was made to the source code material. If not followed strictly, the application may not incorporate the new changes made or generate some errors during runtime.

1. First, you must Build the application. You must ensure that the release directory is where you want to store the RELEASED application (directory where the runnable applications will be built and released into). For optimisation and modularity of the applications, you have these settings in your configuration manager like the figure below (Access by hovering over the build button):

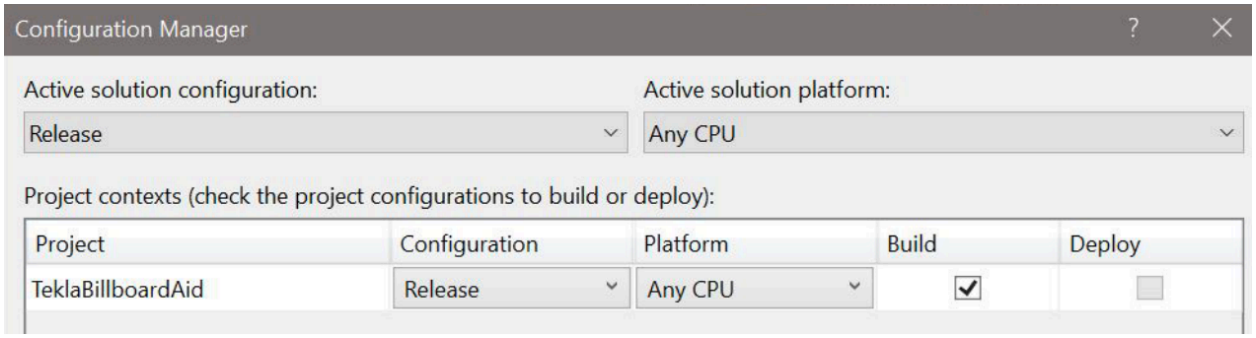


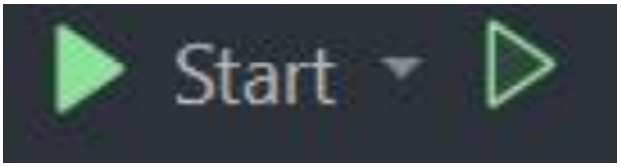
Figure 4: Configuration settings

2. After you have configured the settings, you should either “Clean Solution” then “Build Solution” or “Rebuild” if you have already built it.
3. After pressing this, you must wait and monitor the Output Console for errors. If build is successful, this message should appear on the output console:

```
1>      3 Warning(s)
1>      0 Error(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Build started at 9:53 AM and took 01:01.615 minutes ==========
```

Figure 5: Output console for successful build

4. After successful build, you can manually access the application by searching the directory within the local directory that contains your solution folder. You will need to find **C: …(your local directory)...\ bin\Release.** You can access the newly built application there. You should note that IF YOUR DO NOT FOLLOW THE CONFIGURATION MANAGER SETTINGS, you will need to access the folder that you CHOSE.
5. There is another way of building and running it automatically after building. Use the green buttons on the tool-box above. This will automatically build and open the applications. You can still access the application manually in the directory mentioned in **step 4**. It is recommended that if you want to debug the codes, use this feature instead with breakpoints.



6. The application should appear like this without an error message showing that “***Tekla 2022 not running***“. If it does show, it means that it cannot connect to Tekla Structure software. Usually, this means that the Tekla Structure is not opened – **open it and load either a new or a saved file.**

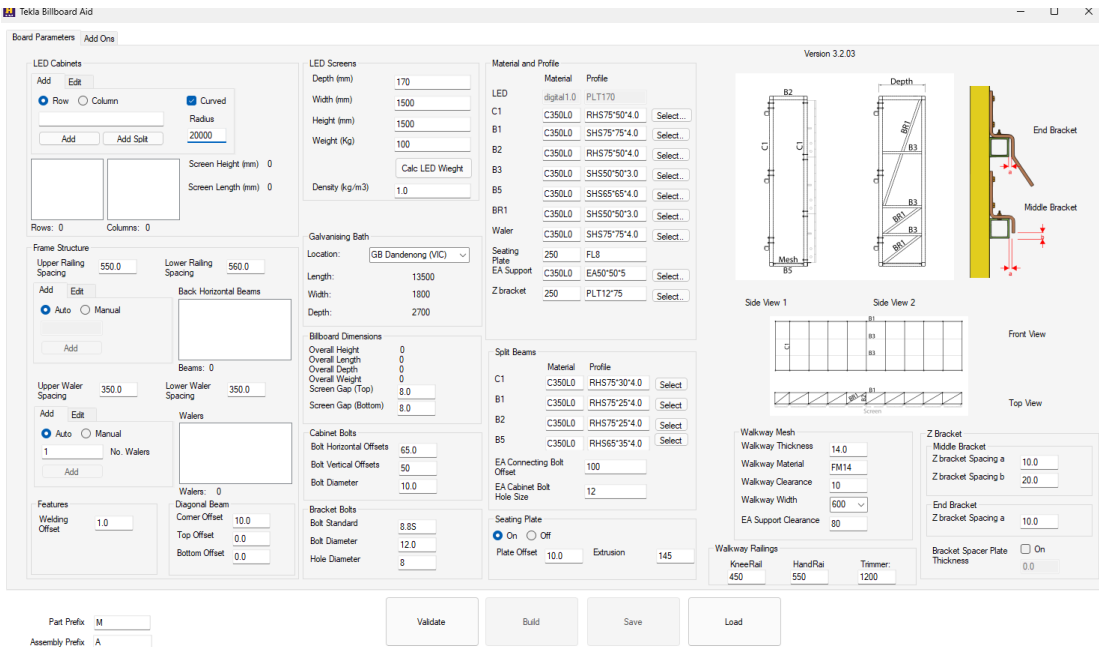


Figure 6: First page of the app

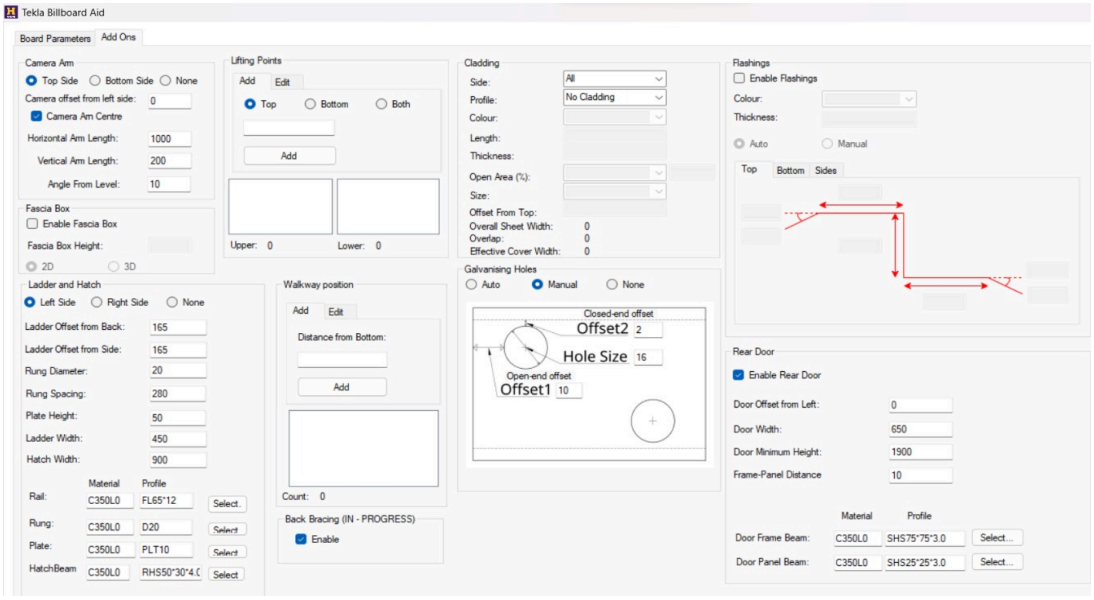


Figure 7: Second page of app

Code Infrastructure

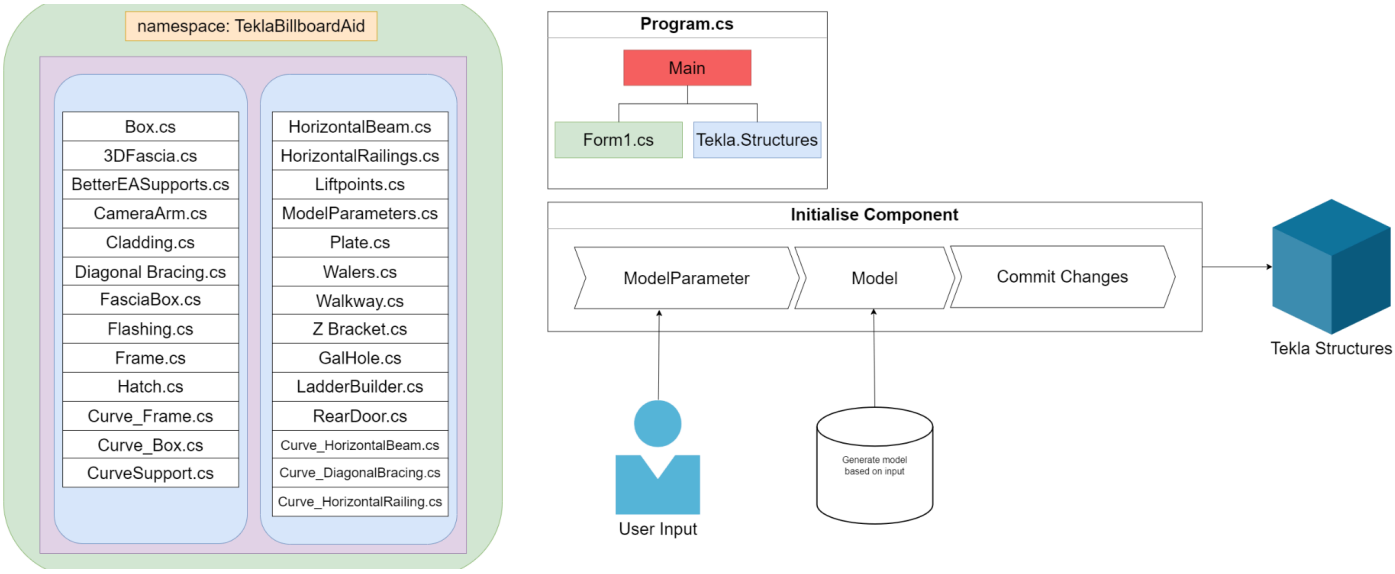


Figure 8: Code Structures of the entire program

The overall code infrastructure incorporates both TeklaOpenAPI plugin template and C# language coding style. All functions/methods are called somewhere within the main code file “Form1.cs” to generate the model inside Tekla Structure.

TeklaOpenAPI Plugin Template: All logic of the application including the interaction with the UI and any logic regarding to getting user inputs and committing changes the “Model” object to the Tekla Structure software is contained in a file named “form1.cs” . This particular file is accompanied by “Form1.cs [Designer]” which allows programmers to visually position the UI elements then is automatically generated into codes by Visual Studio. These two files are the most important and should be modified with precautions.

Namespace: TeklaBillBoardAid: The keyword “namespace” is used to declare the scope of related objects which is extensively used in all files. It is advised that any new changes should be made inside this namespace as different files are dependent on

each other. Any declaration made outside of scope may not provide access for the main program.

All files except “Form1.cs” or any files related to User Interface logic and layout and program flow of the application during run time declare all properties and methods in C# coding standards within the scope of “TeklaBillboardAid” (in other languages with Object Oriented Programming like Java or Python, it can be seen as classes’ attributes and classes’ functions). However there are few objects within the program that are fundamental to the overall infrastructure of model generation within Tekla Structures.

Fundamentals

There are few classes within the program that are the backbone of the entire program when it comes to generating the model as shown in Figure 9 below.

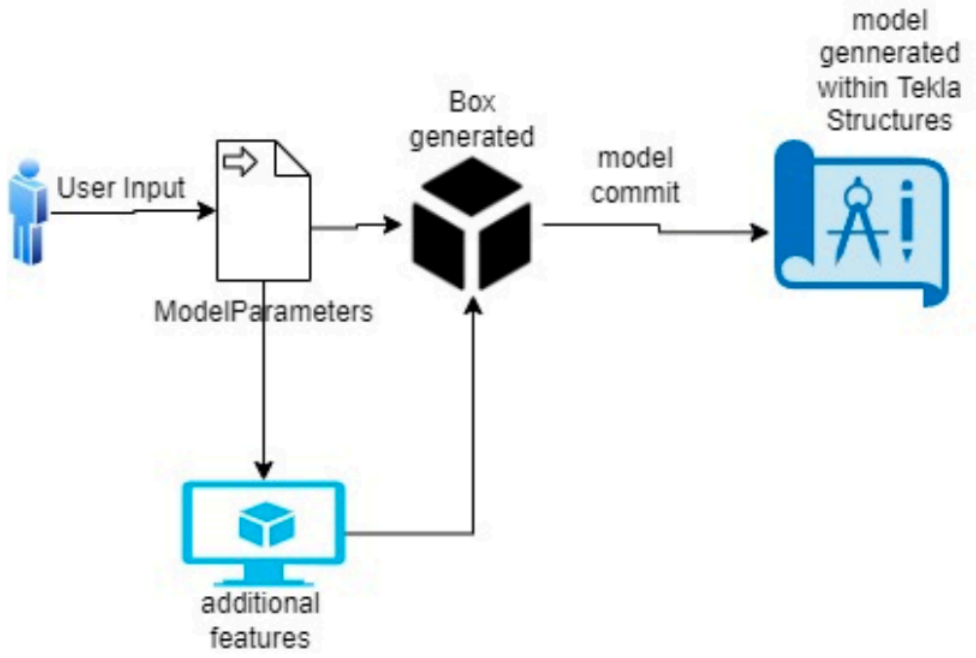


Figure 9: Object position within the program flow

Details of these objects will be detailed in further sections. However, it is established that “box” object, is an entity or assembly within the Tekla Structures (it is defined as a class in TeklaOpenAPI) that within this context, represents either the entire billboard or

a 3D rectangular section of a billboard (if the billboard is split into multiple boxes by enabling splits in the application).
The program mainly adds all necessary features such as frames, bracings and others to the box(es) individually. Other additional features that do not or cannot follow the coding algorithm for box generation are implemented outside of the “box” class declaration scope and rather applied by iteratively modifying each box as it loops through the list of “box” objects. See the next picture for illustrations
of scope of which implementations:

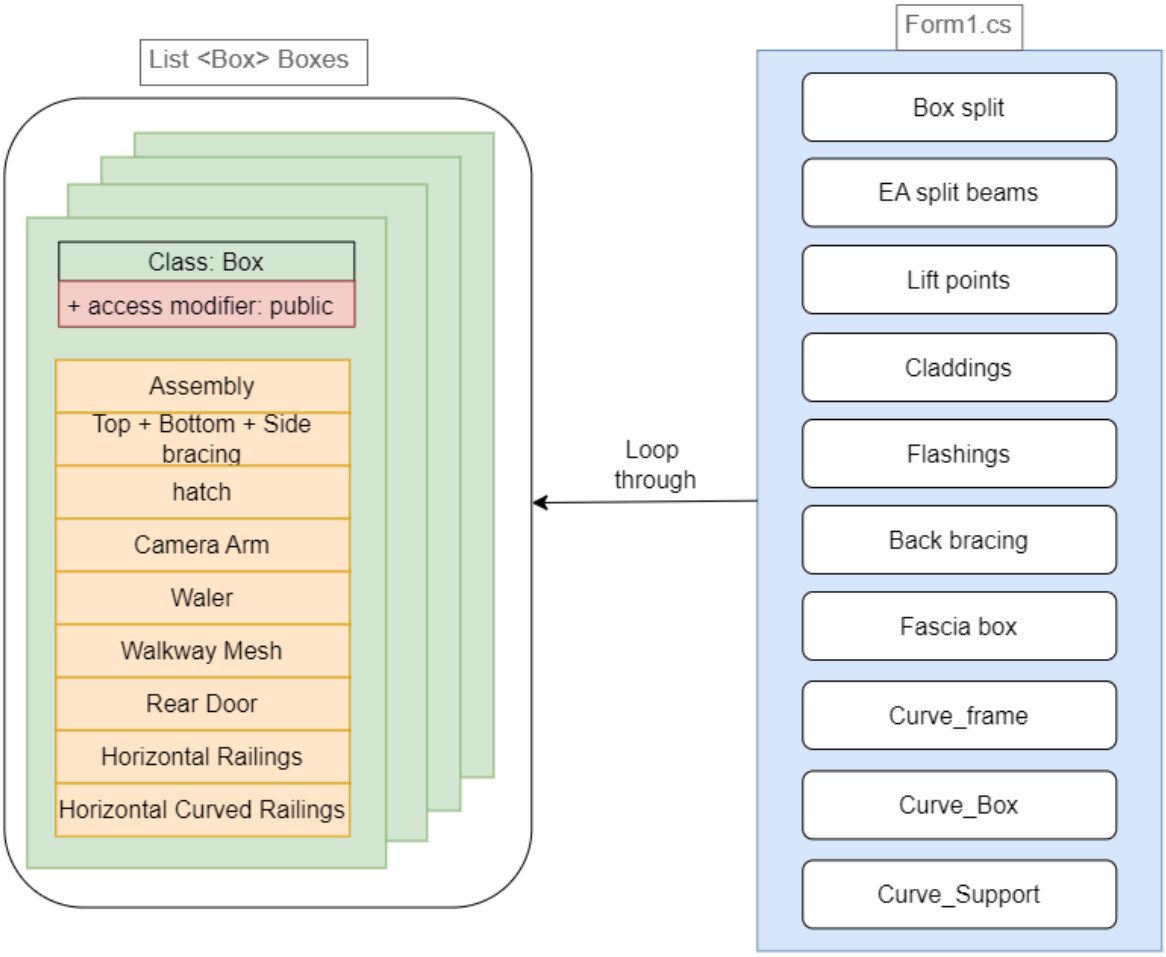


Figure 10: All features of billboard

You can find the final report document to look at how the feature visually generated within Tekla Structures. In this document, we only describe the implementation: class declaration and definition with their properties and methods. Almost all the classes here follow the programming principle of Single-Responsibility Principle which means that any changes to that class should only be done by inherently changing that class by itself, not having an external method or function changing it (**search it up, this is our interpretation).** Note that you may find some inconsistencies with some of the notations as more than 1 person worked on this.

Box Assembly

The box assembly feature groups each of the individual components of a box. It is implemented by modifying the functions that generate the individual component of the box to return a list of Part objects or its subclass, Beam objects. Those lists would be collected at Box.cs when each box is generated. At the end of the box generation, it would use the .GetAssembly() function of Part objects and its child classes with B1Beam1 to obtain the main assembly into which the rest of the components will be added. Then, set B1Beam1 as the main part of the assembly. This code is illustrated in Figure 11. There is no particular reason B1Beam1 is selected for it; it is merely because it is the first one, and it should work with other Part objects or its child classes as the main assembly. With the main assembly established and like the B1Beam1, the rest of the components are added with the addition that the assembly will then be added into the main assembly of the box. There is also a helper function to add a list of Part objects into the assembly; this is demonstrated in Figure 11.

```
// Add the B1 beams into the assembly.
_boxAssembly = B1Beam1.GetAssembly();
_boxAssembly.SetMainPart(B1Beam1);

Assembly B1Beam2Assembly = B1Beam2.GetAssembly();
B1Beam2Assembly.SetMainPart(B1Beam2);

Assembly B1Beam3Assembly = B1Beam3.GetAssembly();
B1Beam3Assembly.SetMainPart(B1Beam3);

Assembly B1Beam4Assembly = B1Beam4.GetAssembly();
B1Beam4Assembly.SetMainPart(B1Beam4);

_boxAssembly.Add(B1Beam2Assembly);
_boxAssembly.Add(B1Beam3Assembly);
_boxAssembly.Add(B1Beam4Assembly);
```

```
8 references
private void AddBeamsToAssembly(List<Part> Parts)
{
    foreach (Part P in Parts)
    {
        Assembly PartAssembly = P.GetAssembly();
        PartAssembly.SetMainPart(P);

        _boxAssembly.Add(PartAssembly);
    }
}
```

Figure 11: Box Assembly

Rear Door

Examples Rear Door in the model looks like following Figure 12.

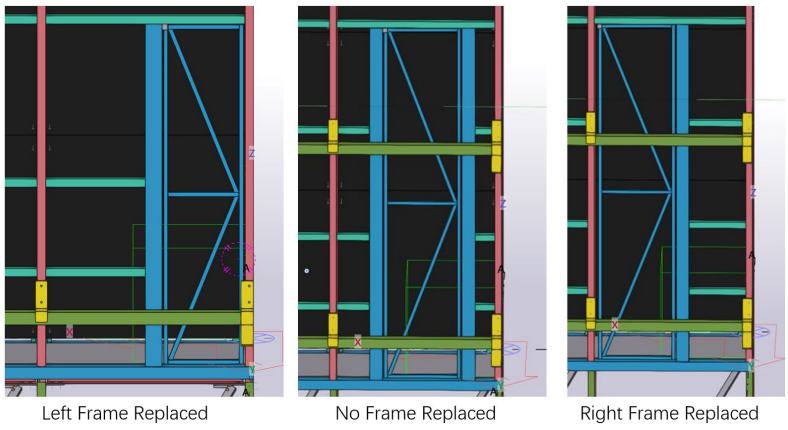


Figure 12: Rear Door Model

Labelled Rear door as followings Figures 13:

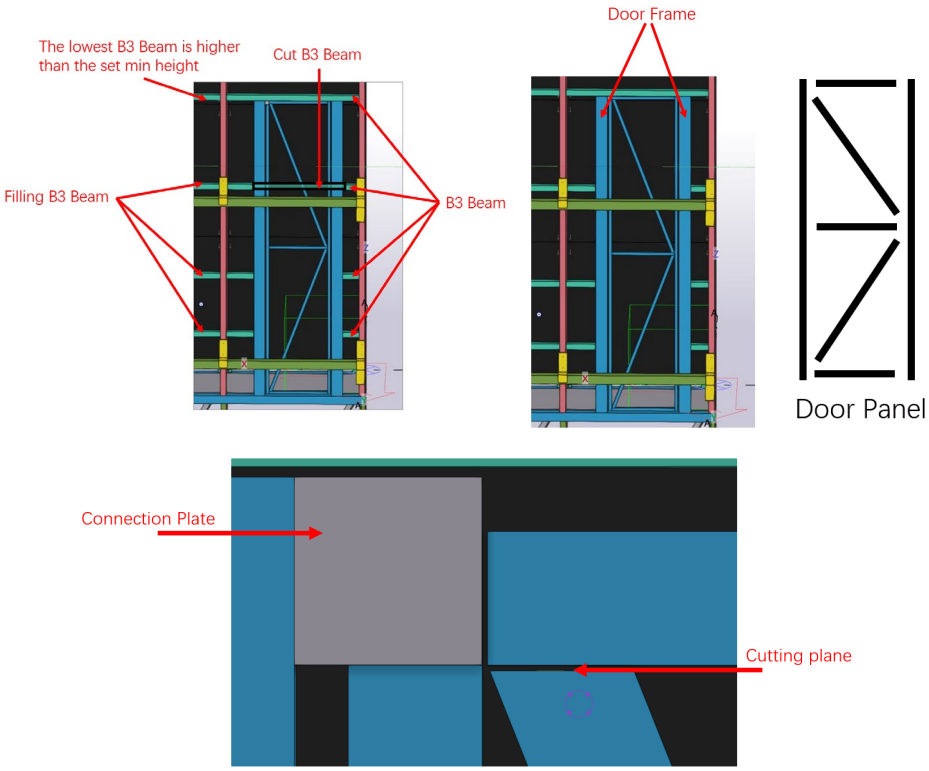


Figure 13: Rear Door Details

The RearDoor class is responsible for creating the rear door components, including door frames, door panels, and bracings. It takes the ModelParameters and other necessary parameters as input. And in order to follow phase 1&2 coding style and box assembly feature, it would generate a rear door in Box.cs according to user input. Shown in Figure 14 ,the door is not enabled when the application start. All other input for the rear door is not allowed and in grey colour.

Figure 14: Rear Door UI

The RearDoor class has a constructor (Figure 15)that takes the necessary parameters, including ModelParameters, boxZ, doorTopZ, beamCutted, leftFrameReplace, and rightFrameReplace.

```
/// <summary>
/// Constructor for RearDoor class.
/// </summary>
/// <param name="modelparameters">The model parameters.</param>
/// <param name="BoxZ">The Z-coordinate of the box.</param>
/// <param name="DoorTopZ">The Z-coordinate of the top of the door.</param>
/// <param name="BeamCutted">The list of cut beams.</param>
/// <param name="LeftFrameReplace">True if the left frame should be replaced.</param>
/// <param name="RightFrameReplace">True if the right frame should be replaced.</param>
1 reference
public RearDoor(
    ModelParameters modelparameters,
    double BoxZ,
    double DoorTopZ,
    List<Beam> BeamCutted,
    bool LeftFrameReplace,
    bool RightFrameReplace)
{
    this.modelparameters = modelparameters;
    this.BoxZ = BoxZ;
    this.DoorTopZ = DoorTopZ;

    this.BeamCutted = BeamCutted;
    this.LeftFrameReplace = LeftFrameReplace;
    this.RightFrameReplace = RightFrameReplace;

}
```

Figure 15: Rear Door Constructor

Shown in Figure 16, DoorFrameBeam(), DoorPanelBeam(), DoorConnection() and FillingBeam() set the basic properties for these beams and would be used in rear door the creation part.

```
2 references
private Beam DoorFrameBeam()
{
    //Set door frame properties
    Beam DoorFrame = new Beam();
    DoorFrame.Name = "DoorFrame";
    DoorFrame.Profile.ProfileString = modelparameters.DoorFrameProfile;
    DoorFrame.Material.MaterialString = modelparameters.DoorFrameMaterial;
    this.DoorFrameBeamWidth = modelparameters.DoorFrameWidth;
    this.DoorFrameBeamHeight = modelparameters.DoorFrameHeight;
    DoorFrame.Class = "11";
    return DoorFrame;
}

7 references
private Beam DoorPanelBeam()
{
    //Set door panel properties
    Beam DoorPanel = new Beam();
    DoorPanel.Name = "DoorPanle";
    DoorPanel.Profile.ProfileString = modelparameters.DoorPanelProfile;
    DoorPanel.Material.MaterialString = modelparameters.DoorPanelMaterial;
    DoorPanelBeamHeight = modelparameters.DoorPanelHeight;
    DoorPanelBeamWidth = modelparameters.DoorPanelWidth;
    DoorPanel.Class = "11";
    return DoorPanel;
}

1 reference
private ContourPlate DoorConnection()
{
    // Set door plate properties
    ContourPlate ConnectionPlate = new ContourPlate();
    ConnectionPlate.Profile.ProfileString = "PLT" + ((DoorFrameBeamWidth - DoorPanelBeamWidth) / 2).ToString();
    ConnectionPlate.Material.MaterialString = "C350L0";
    ConnectionPlate.Class = "1";
    return ConnectionPlate;
}
```

```
// Filling cutted B3 beam
1 reference
private Beam FillingBeam()
{
    // Set filling B3 beam profile
    Beam FillingBeam = new Beam();
    FillingBeam.Name = "Beam";
    FillingBeam.Class = "5";
    FillingBeam.Profile.ProfileString = modelparameters.B3Profile;
    FillingBeam.Material.MaterialString = modelparameters.B3Material;
    return FillingBeam;
}
```

Figure 16: Rear Door Funtaions

The BuildRearDoor method in the RearDoor class is the main method responsible for creating the rear door. It takes StartPointX and EndPointX as input parameters, which define the X-coordinate range of the door. Inside the BuildRearDoor method, the door parts are created step-by-step:

- Door frames (DoorFrameLeft and DoorFrameRight) are created and positioned based on the provided StartPointX and EndPointX. If the leftFrameReplace and

rightFrameReplace flags are set, the door frames are adjusted accordingly. The following Figure 17 is how we create left frame of the door.

```
private List<Part> DoorCreator(double StartPointX, double EndPointX)
{
    List<Part> DoorParts = new List<Part>();
    double DoorFramePanelSpacing = modelparameters.DoorPanelFrameSpacing;

    //Get B3 beam dimensions
    string[] B3Beamparameter = modelparameters.B3Profile.Split('S')[2].Split('*');
    double B3BeamHeight = double.Parse(B3Beamparameter[0]);
    double B3BeamWidth = double.Parse(B3Beamparameter[1]);

    // Create left door frame beam
    Beam DoorFrameLeft = DoorFrameBeam();
    DoorFrameLeft.StartPoint = new TSG.Point(
        StartPointX,
        modelparameters.BillboardDepth - DoorFrameBeamWidth / 2,
        BoxZ  + modelparameters.WeldOffset);

    DoorFrameLeft.EndPoint = new TSG.Point(
        DoorFrameLeft.StartPoint.X,
        DoorFrameLeft.StartPoint.Y ,
        DoorTopZ - B3BeamHeight/2 - modelparameters.WeldOffset);

    // If the left door's left most point is same as the column left most point then use the column as door's left frame
    // Else insert the created left frame beam
    if (!LeftFrameReplace)
    {
        if (!DoorFrameLeft.Insert()) { MessageBox.Show("Insertion of left door frame failed."); }
        DoorParts.Add(DoorFrameLeft);
    }
```

Figure 17: Door Frame Constructor

- Door panels (DoorPanelLeft, DoorPanelRight, DoorPanelTop, DoorPanelBottom, and DoorPanelMiddle) are created and positioned relative to the door frames. The top and bottom bracings (DoorPanelBrcingtop and DoorPanelBracingbottom) are also created for added support. And the code for the panel is similar to the frame. And following Figure 18 shows the code for top bracing in door panel.

```
// Calculate xchange and zchange for the panel bracing startpoint
double alpha = Math.Atan(
    (DoorPanelTop.EndPoint.X - DoorPanelTop.StartPoint.X) /
    (DoorPanelTop.StartPoint.Z - DoorPanelMiddle.StartPoint.Z + DoorPanelBeamHeight));
double diaganolL = Math.Sqrt(
    Math.Pow((DoorPanelTop.EndPoint.X - DoorPanelTop.StartPoint.X), 2) +
    Math.Pow((DoorPanelTop.StartPoint.Z - DoorPanelMiddle.StartPoint.Z + DoorPanelBeamHeight), 2));
double belta = Math.Asin(DoorPanelBeamHeight / diaganolL);
double Xchange = Math.Cos(alpha - belta) * DoorPanelBeamHeight;
double Zchange = Math.Sin(alpha - belta) * DoorPanelBeamHeight;

// Create and insert top bracing in door panel
Beam DoorPanelBrcingtop = DoorPanelBeam();
DoorPanelBrcingtop.StartPoint = new TSG.Point(
    DoorPanelTop.EndPoint.X - Xchange,
    DoorPanelTop.EndPoint.Y ,
    DoorPanelTop.EndPoint.Z - DoorPanelBeamHeight + Zchange );
DoorPanelBrcingtop.EndPoint = new TSG.Point(
    DoorPanelMiddle.StartPoint.X,
    DoorPanelMiddle.StartPoint.Y,
    DoorPanelMiddle.StartPoint.Z + modelparameters.WeldOffset);

if (!DoorPanelBrcingtop.Insert()) { MessageBox.Show("Insertion of door panel bracing failed"); }
DoorParts.Add(DoorPanelBrcingtop);

// Create top cutplane for door panel top bracing
CutPlane CutPlaneToptop = new CutPlane
{
    Plane = new Plane
    {
        Origin = DoorPanelTop.StartPoint - new TSG.Point(0,0,DoorPanelBeamHeight + modelparameters.WeldOffset),
        AxisY = new TSG.Vector(0,modelparameters.BillboardDepth,0),
        AxisX = new TSG.Vector(modelparameters.BillboardHeight,0,0)
    }
};
CutPlaneToptop.Father = DoorPanelBrcingtop;
if (!CutPlaneToptop.Insert()) { MessageBox.Show("Door panel upper bracing cut failed"); }
```

Figure 18: Door Panel Construtor

- The connection plate (DoorConnectionPlate) is created and positioned between the door frames to join them together. This plate only serves an identification function and does not have any practical effect.
- If any B3 beams need to be cut due to the door's height, they are cut using CutPlane objects and filling by FillingBeam(). The cut plane is similar to the cut plane code for bracing.
- Finally, all the created door parts are added to the doorParts list for box assembly in Box.cs and returned.

For future improvements, the cutting beam part can be removed from RearDoor and placed in the appropriate location in Box.cs. This would make RearDoor more modular. Additionally, the RearDoor code is a bit cluttered, and unnecessary input parameters and code can be removed to maintain code cleanliness.

Ladders

Following is the comment Figure 19 for ladder

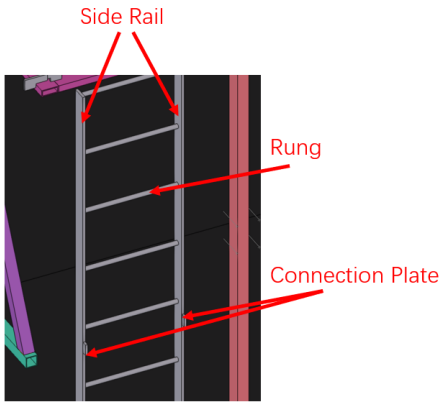


Figure 19: Detailed Ladder Model

The `LadderBuilder.cs` file contains the implementation of the `LadderBuilder` class, responsible for generating ladders based on user inputs and adding them to the Tekla Structures model in Box.cs. Here is the explanation of the code design and logic:

- Class Definition and Constructor(Figure 19):

- The `LadderBuilder` class is defined in the `TeklaBillboardAid` namespace.

- The class has a constructor that takes `ModelParameters` as a parameter. The `ModelParameters` object contains necessary data for ladder construction.

```
2 references
public LadderBuilder(ModelParameters modelparameter)
{
    this.modelParameters = modelparameter;
```

Figure 20: Class Definition and Constructor

- BuildLadder Method (Figure 20):

- The `BuildLadder` method takes a starting point, an ending point, and a list of railing heights to construct the ladder.

- It extracts parameters such as `C1BeamWidth`, `B3BeamHeight`, and `B3BeamWidth` from the `ModelParameters` object to use in the ladder construction.

- The start and end points are adjusted by adding an offset to them to align with the ladder beams.

- The railing heights are adjusted to account for the height of the B3 beam.

- The method then creates a side rail, a rung, and connection plates based on the given inputs and `ModelParameters` data.

- Finally, it connects all the ladder parts together and returns a list of parts representing the ladder.

```
public List<Part> BuildLadder(TSG.Point startpoint, TSG.Point endpoint, List<double> RailingHeights)
{

    // Extracting C1 Beam parameters from the ModelParameters object
    string[] C1Beamparameter = modelParameters.C1Profile.Split('*');
    double C1BeamWidth = double.Parse(C1Beamparameter[1]);

    // Adjusting start and end points by adding an offset
    startpoint.X += (C1BeamWidth/2 );
    endpoint.X += (C1BeamWidth/2);

    // Extracting B3 Beam parameters from the ModelParameters object
    string[] B3Beamparameter = modelParameters.B3Profile.Split('S')[2].Split('*');
    double B3BeamHeight = double.Parse(B3Beamparameter[0]);
    double B3BeamWidth = double.Parse(B3Beamparameter[1]);

    // Adjusting railing heights to account for B3 Beam height
    for (int i = 0; i < RailingHeights.Count; i++)
    {
        RailingHeights[i] -= B3BeamHeight / 2;
    }

    // Adjusting Connection plate Y position
    double ConnectionPlateY = modelParameters.BillboardDepth - B3BeamWidth;

    // Creating side rail, rung, and plates list
    Beam sideRail = CreateSideRail(startpoint,endpoint);
    Beam rung = CreateRung(startpoint);
    List<ContourPlate> plates = CreatePlate(startpoint, RailingHeights,ConnectionPlateY);

    // Inserting the side rail, rung and plate into the model
    List<Part> LadderPart = ConnectLadder(sideRail, rung,plates);
    return LadderPart;
}
```

Figure 20: Ladder Constructor

- CreateSideRail Method (Figure 21):

- This method creates a side rail for the ladder based on the given start and end points.

- It sets properties such as profile, material, name, class, and position for the side rail.

```
75      private Beam CreateSideRail(TSG.Point startpoint, TSG.Point endpoint)
76      {
77          // Setting properties for the side rail
78          Beam SideRail = new Beam();
79          SideRail.Name = "Side Rail";
80          SideRail.Profile.ProfileString = modelParameters.LadderRailProfile;
81          SideRail.Material.MaterialString = modelParameters.LadderRailMaterial;
82          SideRail.Class = "Ladder";
83          SideRail.StartPoint = new TSG.Point(startpoint.X,startpoint.Y,startpoint.Z);
84          SideRail.EndPoint = new TSG.Point(endpoint.X,endpoint.Y,endpoint.Z);
85          SideRail.Position.Rotation = Position.RotationEnum.FRONT;
86          SideRail.Position.Plane = Position.PlaneEnum.MIDDLE;
87          SideRail.Position.Depth = Position.DepthEnum.MIDDLE;
88          SideRail.Class = "1";
89          return SideRail;
90      }
```

Figure 21: Rail Bar Constructor

- CreateRung Method (Figure 22):

- This method creates a rung for the ladder based on the given start point.

- It sets properties such as profile, material, name, class, and position for the rung.

```
97      private Beam CreateRung(TSG.Point startpoint)
98      {
99          // Setting properties for the rung
100         Beam Rung = new Beam();
101         Rung.Name = "Rung";
102
103         Rung.Profile.ProfileString =modelParameters.LadderRungProfile;
104         Rung.Material.MaterialString = modelParameters.LadderRungMaterial;
105         Rung.Class = "Ladder";
106         Rung.Class = "1";
107         Rung.Position.Rotation = Position.RotationEnum.FRONT;
108         Rung.Position.Plane = Position.PlaneEnum.MIDDLE;
109         Rung.Position.Depth = Position.DepthEnum.MIDDLE;
110         return Rung;
111     }
```

Figure 22: Rung Constructor

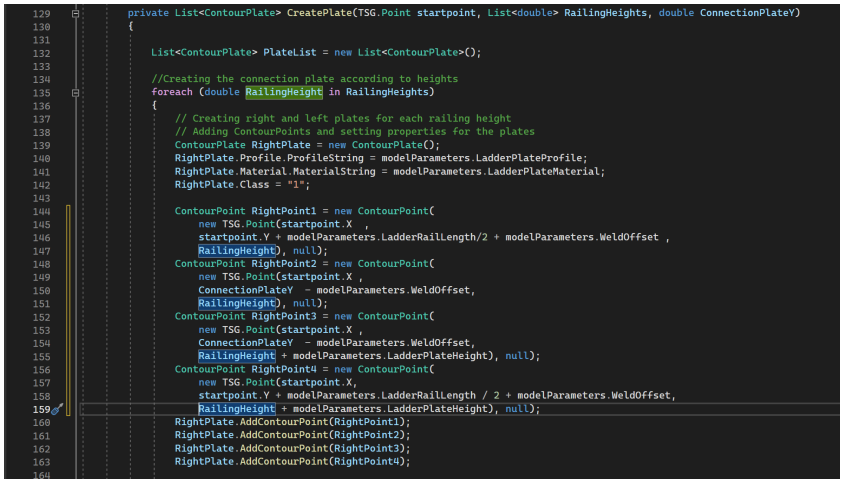
- CreatePlate Method (Figure 23):

- This method creates plates for ladder connections based on the given start point, railing heights, and the `ConnectionPlateY` value.

- It calculates the positions of right and left plates based on the railing heights and connection plate Y-coordinate.

- Each plate is added to the `PlateList`, which is later returned.

- When creating a plate in code, you have to input the coordinates of the four corners in either clockwise or counterclockwise order, just like when creating a plate in Tekla software. If the coordinates are not entered in the correct order, the model will not be generated, but there won't be any error prompts either.



```
129     private List<ContourPlate> CreatePlate(TSG.Point startpoint, List<double> RailingHeights, double ConnectionPlateY)
130     {
131
132         List<ContourPlate> PlateList = new List<ContourPlate>();
133
134         //Creating the connection plate according to heights
135         foreach (double RailingHeight in RailingHeights)
136         {
137             // Creating right and left plates for each railing height
138             // Adding ContourPoints and setting properties for the plates
139             ContourPlate RightPlate = new ContourPlate();
140             RightPlate.Profile.ProfileString = modelParameters.LadderPlateProfile;
141             RightPlate.Material.MaterialString = modelParameters.LadderPlateMaterial;
142             RightPlate.Class = "1";
143
144             ContourPoint RightPoint1 = new ContourPoint(
145                 new TSG.Point(startpoint.X  ,
146                 startpoint.Y + modelParameters.LadderRailLength/2 + modelParameters.WeldOffset ,
147                 RailingHeight), null);
148             ContourPoint RightPoint2 = new ContourPoint(
149                 new TSG.Point(startpoint.X ,
150                 ConnectionPlateY  - modelParameters.WeldOffset,
151                 RailingHeight), null);
152             ContourPoint RightPoint3 = new ContourPoint(
153                 new TSG.Point(startpoint.X ,
154                 ConnectionPlateY  - modelParameters.WeldOffset,
155                 RailingHeight + modelParameters.LadderPlateHeight), null);
156             ContourPoint RightPoint4 = new ContourPoint(
157                 new TSG.Point(startpoint.X,
158                 startpoint.Y + modelParameters.LadderRailLength / 2 + modelParameters.WeldOffset,
159                 RailingHeight + modelParameters.LadderPlateHeight), null);
160             RightPlate.AddContourPoint(RightPoint1);
161             RightPlate.AddContourPoint(RightPoint2);
162             RightPlate.AddContourPoint(RightPoint3);
163             RightPlate.AddContourPoint(RightPoint4);
164
```

Figure 23: Connection Plate Constructor

- ConnectLadder Method:

- This method connects the side rail, rungs, and plates to build the ladder.

- It inserts both right and left side rails into the model from the back view.

- It calculates the number of rungs needed based on the side rail length and inserts them accordingly.

- It inserts the connection plates one by one into the model.

The code follows an object-oriented approach, breaking the ladder-building process into smaller methods for better organization and readability. The `ModelParameters` class provides the necessary data to construct the ladder, making the `LadderBuilder` class flexible and reusable for different scenarios. The ladder parts are inserted into the Tekla Structures model using appropriate Tekla API methods. Moreover, the LadderBuilder code could benefit from refactoring to improve code cleanliness. Unnecessary input parameters and code can be removed to enhance readability and maintainability.

Hatches

The hatches are comprise of 7 beams, 4 hinge plate forming 2 hinges, and a seating plate. The 7 beams include the 5 beams (front, back, left, right, and diagonal) making up the main part of the hatch and 2 support B2 beam (left and right support beam). The hinges (front and back hinges) connectes the support beam and either the left or right beam of the main hatch part. An example hatch model with labelled parts can be seem below.

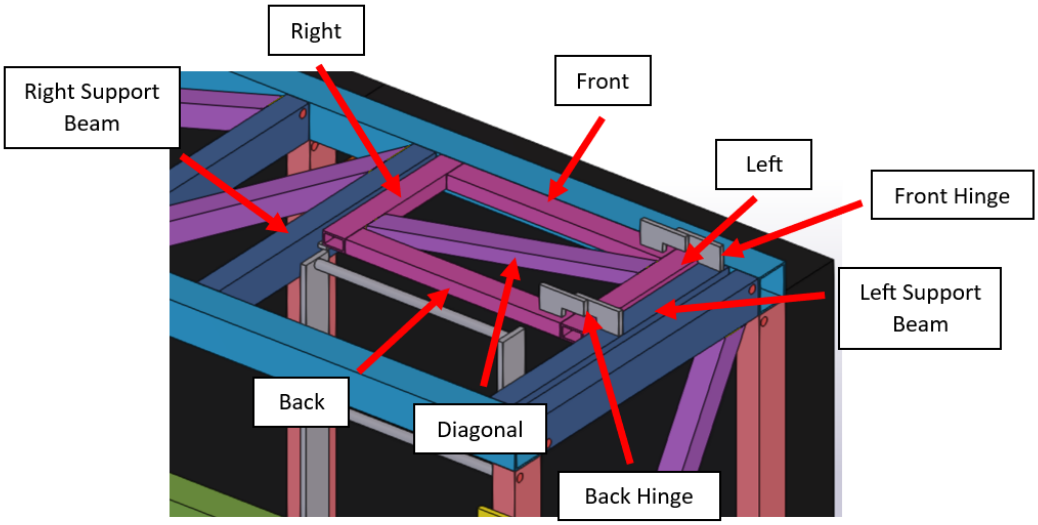


Figure 24: Detailed Hatch Model

The generation of the hatch is written in the Hatch class in Hatch.cs file. The Hatch class takes in HatchStart and HatchEnd points which denotes the diagonal point of the hatch's inner space (i.e. inside the frame). The class also takes HatchZ, which is the Z coordinate of the hatc, YDistanceForLadder, which is the distance between the hatch and the back B1 beam, and IsLeftHatch that indicate whether the hatch is on the left or right side of the billboard. Some of the parameters are illustrated in the figure below. The HatchStart and HatchEnd parameters does not include the 10mm space that will be left out from both the left and right support beam for the hinges. HingeSpace is the distance between the support beams with the hatch main beams, it is created so that the hatch can be opened.

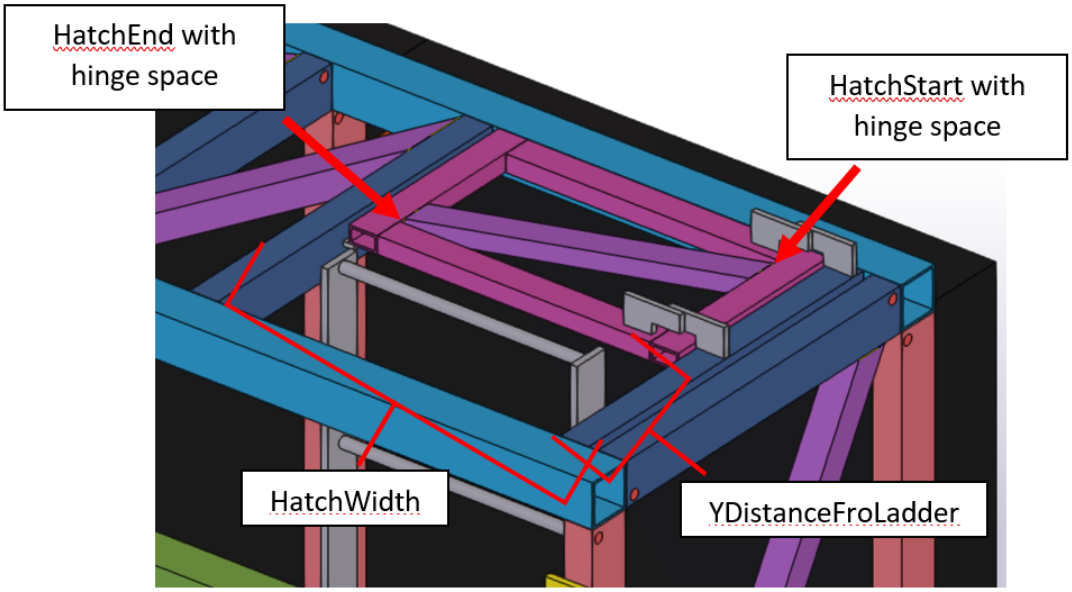


Figure 25: Detailed Hatch Model

The beam creation process is as below:

1. Create the front beam.
2. Create the back beam.
3. Create the left beam.
4. Create the right beam.
5. Create the diagonal beam.
6. Create left B2 beam.
7. Create right B2 beam.
8. Create the hinges.
9. Create the seating plate.

For the hatch to integrate into the billboard, Form1, Box, and Frame classes are also modified. Frame is modified so that the top and bottome frame can generate by choice as the hatches would be there. The Box class is modifed so that it takes in the specification of the Hatch and changes the generation of frames to fit in the hatch. In additional to that, the Box class includes the generation of the hatch including determining the start and end point, and the top and

bottom diagonal bracing generation of the box is modified to adapt the hatch. Form1 is modifed to provide the hatch specification to the box through what was entered in the UI.

Side Bracing

containment - Diagonal Bracing.cs			
Class Definition			
Access modifer	Modifer	Data Type	Name
Public	n/a	Class	Diagonal
Class properties			
Access modifer	Modifer	Data Type	Name
Public	Static	Class	CameraArmLeft
Public	Static	Class	CameraArmRight
Public	Static	Class	CameraArm
Class methods			
Access modifer	Modifer	Return Type	Name
Public	Static	List<Beam>	DiagonalBracing
Public	Static	Beam	CreateDiagonalBeam
Public	Static	List<Beam>	CameraFinder
Public	**Static**	**List<Beam>**	**InstallSideBracing**
Private	**Static**	**List<Beam>**	**BraceGenerate**
Public	Static	Void	BackShot

Table 1: Diagonal Bracing Class

Methods highlighted are used to install bracing on the sides of the bill board. Originally it was combined to be one function but separated to allow better code design and modularity. Both functions are called within **box.cs** file – code snippet below.

```
830             // INSERT SIDE BRACING
831             List<Part> SideBraces = new List<Part>();
832
833             List<double> SideCoords = new List<double>  { 0.0, modelParameters.BillboardLength };
834
835             // Side bracing on LEFT side
836             SideBraces.AddRange(Diagonal.InstallSideBracing(HorizontalRailingsBeams,
837             modelParameters,
838             boxLength,
839             boxHeight,
840             OriginOffset,
841             SideCoords[0],
842             side1,
843             side3
844             ));
845
846             // Side bracing on RIGHT side
847             SideBraces.AddRange(Diagonal.InstallSideBracing(HorizontalRailingsBeams,
848             modelParameters,
849             boxLength,
850             boxHeight,
851             OriginOffset,
852             SideCoords[1],
853             side1,
854             side3
855             ));
```

Parameters for InstallSideBracing()		
Input Parameters	**Type**	**Description**
HorizontalRailings	List <Class - Part>	List of <PART> objects representing all horizontal railings installed within the box.
modelParameters	Class < ModelParameters>	Class storing all inputed data to generate model
BoxLength	double	Length of box
BoxHeight	double	Height of box
BoxOriginPos	Class <TSG.POINT>	Origin position of the box in (x,y,z)
SideCoords	double	Coordinates of side of the bill board on X-axis
BottomSplit	boolean	Boolean representing if top of box is a split (True = Split)
TopSplit	boolean	Boolean representing if bottom of box is a split (True = Split)
Output Parameters	**Type**	**Description**
SideBraces	List <Class - Beam>	List of side braces generated used to add to box assembly

Table 2: Parameters for InstallSideBracing()

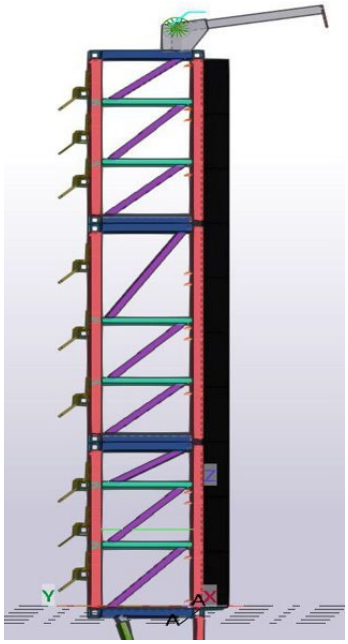
Parameters for BraceGenerate()		
Input Parameters	**Type**	**Description**
modelParameters	Class < ModelParameters>	Class storing all inputed data to generate model
BoxLength	double	Length of box
BoxHeight	double	Height of box
BoxOriginPos	Class <TSG.POINT>	Origin position of the box in (x,y,z)
Side	double	Coordinates of side of the bill board on X-axis
BottomSplit	boolean	Boolean representing if top of box is a split (True = Split)
TopSplit	boolean	Boolean representing if bottom of box is a split (True = Split)
Output Parameters	**Type**	**Description**
SideBraces2Return	List <Class - Beam>	List of side braces generated used to add to box assembly

Table 3: Parameters for BraceGenerate()

Installation of side braces is implemented in the object "box" constructor. Function "InstallSideBracing()" is called twice within the constructor for left and right side of the bill board. To prevent accidental side bracing generation for every box, the function compares the left both left and right side coordinates of the "box" object to the left and right sides of the entire bill board.

```
2 references
787         private static List<Beam> BraceGenerate(List<Double> ZSpacing,
788             ModelParameters modelParameters,
789             double BoxLength,
790             double BoxHeight,
791             TSG.Point BoxOriginPos,
792             double Side,
793             bool BottomSplit,
794             bool TopSplit
795             )
796         {
```

If either matches, function BraceGenerate is called to install the bracings for that side and box. Within the "BraceGenerate()" function, all work done to generate braces including searching starting and endpoint of the beam and inserting cutting planes are included. The function first set the starting and ending coordinates then offsets the ending coordinates to align the beams. After that, it insert the cutplanes to prevent the braces going through the horizontal railings on the side. See picture to the right for illustrations.



Offsets on the corner (on y-axis) is customisable and can be set in the first page of the application. Remember that for all welded components, there must be a 1mm gap between the joined component to allow welding offset.

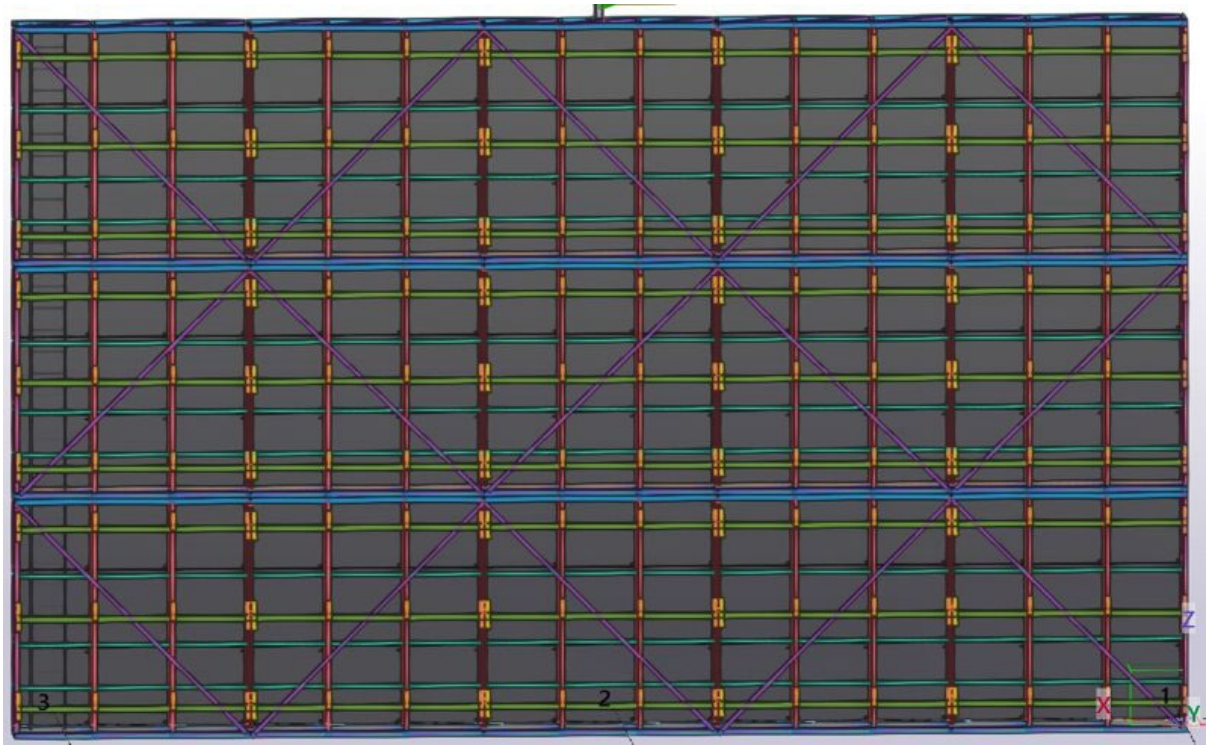
Back Bracing

containment - Diagonal Bracing.cs			
Class Definition			
Access modifer	Modifer	Data Type	Name
Public	n/a	Class	Diagonal
Class properties			
Access modifer	Modifer	Data Type	Name
Public	Static	Class	CameraArmLeft
Public	Static	Class	CameraArmRight
Public	Static	Class	CameraArm
Class methods			
Access modifer	Modifer	Return Type	Name
Public	Static	List<Beam>	DiagonalBracing
Public	Static	Beam	CreateDiagonalBeam
Public	Static	List<Beam>	CameraFinder
Public	Static	List<Beam>	InstallSideBracing
Private	Static	List<Beam>	BraceGenerate
Public	**Static**	**Void**	**BackShot**

Table 4: Diagonal Bracing Class

The highlighted function generates the back bracings for the billboard and is called in Form1.cs rather than in boxes as it has to iterate through the list of boxes and installing a diagonal brace per box with repeating orientation.

For every “box” within the list that as an even index, the starting position is bottom left and ending position is top right where as all other odd indexed “box” as the orientation switched around.



<u>Disclaimer:</u> Installing back braces are quite complex as it depends on the structural loads of the bill boards. Hence, there is no standard style of back bracing. In the future, it is recommended that the style of backbraces should have accommodate more options for user to choose as well as incorporating structural dependent algorithm to install appropriate back bracing.

Note that this feature is incomplete as any beams that intersect diagonal bracing are not properly cut and will require cut planes in the future. As a result, we included in the second page of the application to allow enable/disable the feature (it is enabled by default).

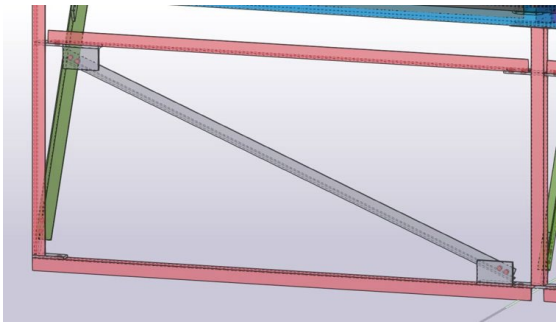
2D Fascia Box Bracing

containment - FasciaBox.cs			
Class Definition			
Access modifer	Modifer	Data Type	Name
Public	n/a	Class	FasciaBox
Class properties			
Access modifer	Modifer	Data Type	Name
Public	n/a	double	VertOffset
Public	n/a	double	HorizOffset
Public	n/a	double	OriginVertOffset;
Public	n/a	Class	p
Class methods			
Access modifer	Modifer	Return Type	Name
Public	n/a	Void	AddEdgeBoltsBolts
Public	n/a	Void	AddColumnBolts
Public	n/a	Void	InsertAngledSupport
Public	**n/a**	**Void**	**InstallFascialBrace**
Private	**n/a**	**List<Double>**	**CalculatePlateDimensions**
Private	**n/a**	**Void**	**fasciaBraceBolt**
Private	**n/a**	**Void**	**CheckFasciaDim**

Table 5: FasciaBox Class

Tables made for each highlighted function are not included as we don't to overcrowd this report . Please look at the source code for more details

2D fascia box bracings come with more complex installation process with a simpler process of attaining starting and end position. Unlike back and side bracings, the bracings used profile of the Equal Angle beams (the beam is L plate beam) and are installed per section of the fascia with the same orientation. Instead of welding, they are bolted to a plate which is welded to the connection plate between the horizontal beams of the fascia box.



The program first generates the bracings then install the bolt and welded plates at both ends of the bracing with a variable bolt plate dimensions depending on the angle of the bracings. Then the plates are bolted while other parts are welded as required by the client. Also, note that if the dimension of the fascia box is too small, the bolts may not accurately place itself within the plates.

Curved Box Assembly

Note: Tables made for each highlighted function are not included as we don't overcrowd this report . Please look at the source code for more details

The Curve_Box class is responsible for bringing together all the components to construct a complete curved box section. It takes as input the column spacings, box height, origin point and other parameters. It then calculates the precise geometry to create the vertical frame on each column using the Curve_Frame class. The frames consist of front, back, top and bottom beams along with seating plates. Additionally, Curve_Box generates the horizontal parts like walers, railings, bracings and brackets at appropriate locations along the curve. Separate classes like Waler, Curve_HorizontalRailings etc. are utilized to create these parts. All the generated beams, plates and other parts are collected into lists. Finally, all part lists are added into a common assembly where the first B1 beam is set as the main part. This groups all individual components into a single box assembly object. So in essence, Curve_Box orchestrates the creation of all parts needed for a curved box segment by using specialized classes for each type of component. It then assembles everything together into a logical assembly for easier manipulation. The modular and hierarchical nature of the implementation allows high extensibility and reusability.

containment - Curve_Box.cs			
Class Definition			
Access modifier	Modifier	Data Type	Name
public	n/a	Class	Curve_Box
Class properties			
Access modifier	Modifier	Data Type	Name
private	n/a	Assembly	_boxAssembly
Class methods			
Access modifier	Modifier	Return Type	Name
private	n/a	Void	AddBeamsToAssembly

Table 6: Curved Box class

Curve Support

The CurveSupport class encapsulates reusable logic to assist with programmatic creation of curved beams and railings in the Tekla Structures model. It has methods like StraightBeam and CurveBeam that handle creating and inserting straight and polycurve beam objects by setting various properties like position, profile, material etc. based on input parameters. To aid in coordinate calculations for curved components, it includes helper methods like Circle_Ycoord and Circle_Xcoord to determine coordinates of points along a circular path. It also has GetCumulativeSum to calculate cumulative lengths along curves which is useful for positioning beams. For creation of horizontal railings, CurveSupport includes methods to calculate z-coordinate points and x-coordinate point pairs along the length where railings need to be placed. The class takes in the overall model dimensions and parameters like spacing, offsets etc. and outputs the created beam and railing objects inserted into the model at appropriate

locations. By encapsulating these repetitive tasks into reusable methods, it simplifies the process of generating the curved geometry and railings programmatically. The orchestrating code then just needs to invoke these methods by passing the right parameters and dimensions for each curved section rather than worrying about the underlying object creation logic.

```
/// <summary> A function to calculate y coordinate of a point on a circle
82 references
public static double Circle_Ycoord(double X_coord, double Circle_Radius, ModelParameters modelParameters)
{
    double Radius = modelParameters.Radius;
    double Width = modelParameters.ScreenLength;

    double Ycoord = -Math.Sqrt(Math.Pow(Circle_Radius, 2) - Math.Pow(X_coord - (Width / 2), 2))
                    + Math.Sqrt(Math.Pow(Radius, 2) - (Math.Pow(Width, 2) / 4));
    return Ycoord;
}

/// <summary> A function to calculate x coordinate of a point on a circle that p ...
99+ references
public static double Circle_Xcoord(double X_coord, double Circle_Radius, ModelParameters modelParameters)
{
    double Radius = modelParameters.Radius;
    double Width = modelParameters.ScreenLength;
    double Angle = Math.Asin((-(Width / 2) + X_coord) / Radius);

    double Xcoord = ((Width * Math.Pow(Math.Tan(Angle), 2)) + Width + 2 * Circle_Radius * Math.Tan(Angle) * Math.Sqrt(Math.Pow(Math.Tan(Angle), 2) + 1))
                    / (2 * (Math.Pow(Math.Tan(Angle), 2) + 1));
    return Xcoord;
}
```

containment - CurveSupport.cs			
Class methods			
Access modifier	Modifier	Return Type	Name
public	static	Beam	StraightBeam
public	static	PolyBeam	CurveBeam
public	static	double	Circle_Ycoord
public	static	double	Circle_Xcoord
public	static	List<double>	HorizontalRailings_Points
public	static	List<(double,	IndividualRailings_Po

		double)>	intsPair
public	static	List<double>	GetCumulativeSum

Table 7: Curved Support class

Parameters for Circle_Ycoord		
Input Parameters	Type	Description
X_coord	double	A double indicating the x coordinate of the point previously on a line
Circle_Radius	double	A double indicating the radius of the circle
modelParameters	ModelParameters	A class storing the parameters of the model
Output Parameters	Type	Description
X	double	a double indicating the x coordinate of the point on a circle
Parameters for Circle_Xcoord		
Input Parameters	Type	Description
X_coord	double	A double indicating the x coordinate of the point previously on a line
Circle_Radius	double	A double indicating the radius of the circle
modelParameters	ModelParameters	A class storing the

		parameters of the model
Output Parameters	Type	Description
X	double	a double indicating the x coordinate of the point on a circle

Table 8: Parameters for Circle_Ycoord and Circle_Xcoord

Curve Frame

The Curve_Frame class contains the core logic to generate the structural frames that make up each curved section of the billboard. It handles building all the beams, plates and other parts needed for an individual frame. The class takes in the 4 corner points, plane type, offsets and other parameters like material properties. It then calculates the precise positions and orientations for the front, back, top and bottom beams based on the radius of curvature and dimensions. Helper methods from the CurveSupport class assist in coordinate calculations along the circular arc. The appropriate profile, class and other properties are assigned to each beam before insertion into the Tekla model. Additionally, Curve_Frame can generate the seating plates under the frames if enabled. It determines the plate contour coordinates based on beam reference points and dimensions. The end plates are defined differently from the middle ones. Finally, the seating plate is inserted after setting all its properties.

containment - Curve_Frame.cs			
Class Definition			
Access modifier	Modifier	Data Type	Name
public	n/a	Class	Curve_Frame
Class properties			

Access modifier	Modifier	Data Type	Name
public	n/a	TSG.Point	Point1
public	n/a	TSG.Point	Point2
public	n/a	TSG.Point	Point3
public	n/a	TSG.Point	Point4
public	n/a	int	PlaneType
public	n/a	Beam	Front
public	n/a	Beam	Back
public	n/a	Beam	Top
public	n/a	Beam	Bottom
public	n/a	ContourPlate	Seatplate
public	n/a	ModelParameters	ModelParameters
public	n/a	bool	SeatingPlateRight
public	n/a	PolyBeam	CutBeamBottom
public	n/a	PolyBeam	CutBeamTop
public	n/a	bool	Side1
public	n/a	bool	Side3
public	n/a	List<Beam>	B2Beams
public	n/a	bool	EnableTopBeam
public	n/a	bool	EnableBottomBeam
Class methods			

Access modifier	Modifier	Return Type	Name
private	n/a	Void	BuildFrame
private	static	ContourPlate	CreateSeatingPlate

Table 9: Curve_Frame class

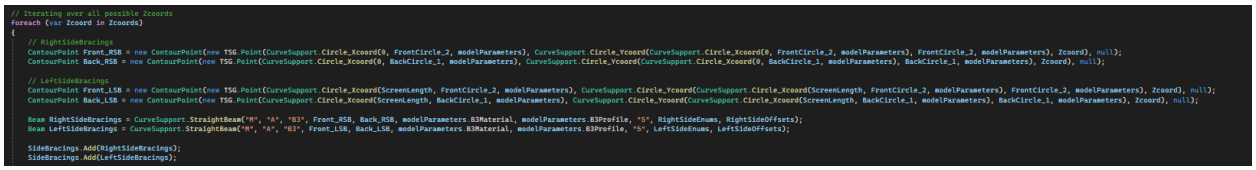
Parameters for CreateSeatingPlate		
Input Parameters	Type	Description
C1BeamOrigin	TSG.Point	A point of the column bottom where the plates should be inserted
frontDimensions	double[]	A list indicating the front dimensions of the column
modelParameters	ModelParameters	A class storing the parameters of the model
EndPlate	bool	A bool to check if Its the first or the last plates for the BillBoard
Output Parameters	Type	Description
CP	ContourPlate	The plates used to make the seating plates for each frame

Table 10: Parameters for CreatreSeatingPlate

Curve Horizontal Railings

The Curve_HorizontalRailings class handles creation of the horizontal railing beams and side bracings along the curve. It takes the model parameters as input and calculates precise positions along the circular arc for each railing and bracing beam to

be inserted. The class first determines all the required z-coordinates and x-coordinate pairs representing start and end points of individual railing segments based on spacing rules and dimensions. It then loops through each z coordinate to create the side bracing beams connecting front to back column at that height. Similarly, for each z, it iterates through the x-coordinate pairs to generate polycurve beams representing horizontal railing segments. Appropriate positions, extends, profiles and other properties are assigned to each beam before insertion into the Tekla model. Finally the class returns two lists containing the created side bracing and horizontal railing beams. By encapsulating this creation logic, the orchestrating Curve_Box class can simply invoke this method without worrying about underlying beam generation mechanics to construct the horizontal members.



containment - Curve_HorizontalRailings.cs			
Class methods			
Access modifier	Modifier	Return Type	Name
public	static	(List<Beam>, List<PolyBeam>)	CurveHorizontalRailings

Table 11: Curve Horizontal Railings Class

Parameters for CurveHorizontalRailings		
Input Parameters	Type	Description
modelParameters	ModelParameters	A class storing the parameters of the model
Output Parameters	Type	Description
SideBracings	List<Beam>	List of Beam objects created for the Both Right and Left Side Bracings
HorizontalRailings	List<PolyBeam>	List of PolyBeam objects created for all the Horizontal Back Railings

Table 12: Parameters for CurveHorizontalRailings

Curve Horizontal Beams

The Curve_HorizontalBeam class is responsible for creating the horizontal B1 beams that run along the curve at the top and bottom of each box section. It takes in the model parameters and calculates precise positions along the circular arc to place the start, mid and end points of each polycurve beam. Appropriate radii are used to determine the coordinates for the front and back B1 beams based on beam depths and offsets. These points are then used to create the four B1 polybeams - two each along the front and back curve. The class assigns required beam properties like profile, class etc. and returns the beams. By encapsulating this horizontal beam creation logic, the calling Curve_Box class simply needs to invoke this method without having to worry about the underlying coordinate transformations and beam insertion tasks. The modularity enables easily changing just this part of the code if the horizontal beam specifications need to be altered.

containment - Curve_HorizontalBeam.cs			
Class methods			
Access modifier	Modifier	Return Type	Name
public	static	(PolyBeam, PolyBeam, PolyBeam, PolyBeam)	CurveHorizontalBeams

Table 13: Curve HorizontalBeam class

Parameters for CurveHorizontalBeams		
Input Parameters	Type	Description
modelParameters	ModelParameters	A class storing the parameters of the model
Output Parameters	Type	Description
B1Beam1	PolyBeam	Return a Bottom Front Circle PolyBeam for the Main Frame
B1Beam2	PolyBeam	Return a Bottom Back Circle PolyBeam for the Main Frame
B1Beam3	PolyBeam	Return a Top Front Circle PolyBeam for the Main Frame
B1Beam4	PolyBeam	Return a Top Back Circle PolyBeam for the Main Frame

Table 14: Parameters for CurveHorizontalBeams

Curve Diagonal Bracing

The CurveDiagonal class is responsible for generating the diagonal bracing beams between front and back faces of the curved billboard structure. It takes in the model parameters and calculates precise start and end points for each diagonal beam to be inserted along the circular profile.The class first determines the required z-coordinates and x-coordinates representing vertical and horizontal bracing locations based on spacing rules. It then loops through the z and x values in pairs to create diagonal beams connecting the front curve to the back curve. Separate left and right bracings are created. Additionally, top and bottom diagonal braces are generated by iterating through x-coordinate ranges. The CreateDiagonalBeam method assigns beam properties like profile, class etc. before insertion. It also adds fittings on both end points to enable connections. Finally, the class returns four lists containing top, left, right and bottom diagonal beams.

containment - Curve_DiagonalBracing.cs			
Class methods			
Access modifier	Modifier	Return Type	Name
public	static	(List<Beam>, List<Beam>, List<Beam>, List<Beam>)	DiagonalBracing
private	static	Beam	CreateDiagonalBeam

Table 15: Curve Diagonal Bracing class

Parameters for DiagonalBracing		
Input Parameters	Type	Description

modelParameters	ModelParameters	A class storing the parameters of the model
Output Parameters	Type	Description
TopDiagonalBRACING	List<Beam>	Return the top diagonal bracing of the BillBoard
LeftDiagonalBRACING	List<Beam>	Return the left side diagonal bracing of the BillBoard
RightDiagonalBRACING	List<Beam>	Return the right side diagonal bracing of the BillBoard
BottomDiagonalBRACING	List<Beam>	Return the bottom diagonal bracing of the BillBoard

Table 16: Parameters for DiagonalBracing

Curved Billboard Mathematics (Desmos)

Desmos was used to plot points with equations that were used in generating a curved billboard in Tekla Structures. Below is the link to the Desmos workspace used for Phase IV:
https://www.desmos.com/calculator/ylyai9ystn?fbclid=IwAR2JPOYI0qvSheEpc-aB027p2cdbzPV_qhAJA7L7pVzItBnP_0V1U_M1cRg

Testing

Comprehensive testing was conducted during this phase to validate the curved modeling capabilities of the system. Test models were generated with different curve radii, dimensions, spacings and component properties. The outputs were thoroughly examined within Tekla to ensure:

- Precise geometry and positioning of all structural members along the circular profile based on parameters
- Accurate creation of box assemblies with appropriate nesting of parts
- Correct orientation and connection fittings for diagonal bracings
- Seamless joining of straight sections to curved extremities
- Conformance to design spacing rules for railings, bracings and other members
- Appropriate mapping of properties like materials, profiles and classes to each component
- Logical grouping of parts into parent assemblies for easier manipulation

Additionally, the modular implementation was validated by tweaking specific classes in isolation and verifying that changes correctly manifested in the integrated model output, without side effects. This confirms loosely coupled code with focused responsibilities.

Test models were also stress analyzed in Tekla for stability. Where issues were noticed during testing, refinements were made iteratively to code or data until outputs conformed to specifications in all aspects. The system can now handle curved billboard design, modeling and customization as intended to significantly enhance capabilities over the initial version.