

Lab1 – 角柱の作成

March 2010 by M. Harada
Updated by Ryuji Ogasawara
Last modified: 5/30/2024

<C#>C#バージョン</C#>

目的:この実習では、ファミリ API の基本を学習していきます。学習する項目は次のとおりです。

- ファミリ環境をチェックする
- 押し出しを使用して単純なソリッドを作成する
- 位置合わせを設定する
- タイプを加える

タスク:長方形のプロファイルを持つ柱ファミリを作成するコマンドを定義して、異なる寸法のバリエーションを持つ 3 つのタイプを加えます:

- (0) テンプレートとして「柱(メートル単位).rft」を使用するものとし、ユーザがこのテンプレートを開いていると仮定する。コマンド内では、念のため、ユーザが正しいテンプレートを選んだかどうかチェックする
- (1) 長方形のプロファイルを定義して押し出し、単純なボックス ソリッドを作成する
- (2) 各面と対応する参照面の間に位置合わせを追加する
- (3) 寸法のバリエーションでタイプを定義する。

図 1 は、この実習に定義する角柱のイメージを示します。

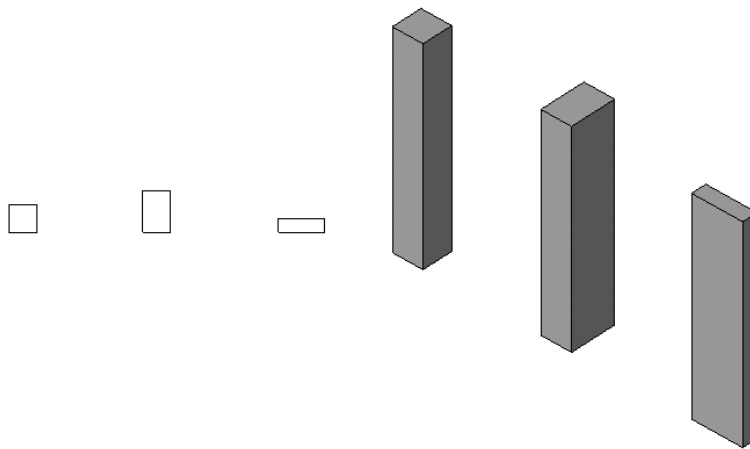
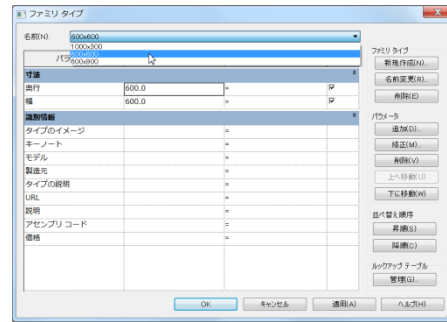
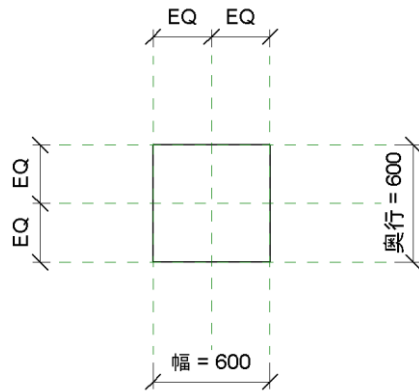


図 1.Lab1 で作成する長方形のプロファイルを持った柱ファミリ

この実習の実装と確認の手順は、下記のとおりです:

1. 外部コマンドを定義する
2. ドキュメント コンテキストの有効性をチェックする
3. 押し出しで単純なソリッドを作成する
4. 位置合わせを追加する
5. タイプを追加する
6. 作成した柱ファミリをテストする

付録 A. Lab1 で使われるヘルパー関数

付録 B.C# のメッセージボックスを表示するヘルパー関数

注意:C#バージョンでは、MessageBox の表示を単純化するために Util クラスを使用します。コードはこのドキュメントの終わりに記述されています。

1. 外部コマンドを定義する

1.1 Visual Studio で開発言語(C#)を選択して新しいクラス ライブラリ(.NET Framework)プロジェクトを作成して、参照を追加して外部コマンドを定義します。ファイル名とクラス名などは、ここでは C#を選択して下記のようにしてください:

- ソリューション名: **FamilyLabs**
- プロジェクト名称: **FamilyLabsCS**
- ファイル名: **1 FamilyCreateColumnRectangle.cs**
- コマンド クラス名:**RvtCmd_FamilyCreateColumnRectangle**
- フレームワーク: **.NET Framework 4.8**

(ここで希望する名前を使用しても構いません。ただし、その場合、プロジェクト名など、このドキュメント内では記述されている名称は、自分でつけた名称で代替して参照してください)

次の参照が少なくとも必要となります:

- System
- System.Core(LINQ クエリ用)
- System.Windows.Forms
- Revit API
- RevitAPIUI

下記はこの実習に必要な名前空間のリストです。それらを.cs ファイルの冒頭に加えてください。

```

<C#>
using System;
using System.Collections.Generic;
using System.Linq; // in System.Core
using Autodesk.Revit;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.ApplicationServices;
</C#>

```

1.2 DB レベルのアプリケーションとドキュメントをそれぞれ保持する m_rvtApp と m_rvtDoc を定義します。下記はその例です:

```

<C#>
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
class RvtCmd_FamilyCreateColumnRectangle : IExternalCommand
{
    // member variables for top level access to the Revit database
    //
    Application _rvtApp;
    Document _rvtDoc;

    // command main
    //
    public Result Execute(
        ExternalCommandData commandData,
        ref string message,
        ElementSet elements )
    {
        // objects for the top level access
        //
        _rvtApp = commandData.Application.Application;
        _rvtDoc = commandData.Application.ActiveUIDocument.Document;

        return Result.Succeeded;
    }
}
</C#>

```

2. ドキュメント・コンテキストの有効性をチェックする

ファミリ API を利用するコマンドは、ファミリ エディタのコンテキスト内でのみで動作します。まず、現在開いているドキュメント コンテキストの有効性をチェックします。このチェックには、isRightTemplate()関数を定義して利用します。isRightTemplate()関数は、引数として BuiltInCategory オブジェクトを受け取ります。

2.1 クラスに次の関数を追加します:

```
<C#>
// =====
// (0) check if we have a correct template
// =====
bool isRightTemplate( BuiltInCategory targetCategory )
{
    // This command works in the context of family editor only.
    //
    if( !_rvtDoc.IsFamilyDocument )
    {
        Util.ErrorMsg( "This command works only in the family editor." );
        return false;
    }

    // Check the template for an appropriate category here if needed.
    //
    Category cat = _rvtDoc.Settings.Categories.get_Item( targetCategory );
    if( _rvtDoc.OwnerFamily == null )
    {
        Util.ErrorMsg( "This command only works in the family context." );
        return false;
    }
    if( !cat.Id.Equals( _rvtDoc.OwnerFamily.FamilyCategory.Id ) )
    {
        Util.ErrorMsg( "Category of this family document does not match the
context required by this command." );
        return false;
    }

    // if we come here, we should have a right one.
    return true;
}
</C#>
```

この関数は、次をチェックします:

- ファミリ ドキュメントかどうか。_rvtDoc.IsFamilyDocument でこれをチェックできます。
- テンプレートが柱ファミリ定義用かどうか。現在のドキュメントのカテゴリは _rvtDoc.OwnerFamily.FamilyCategory.Id を見て確認することができます。

上記の条件が満たされる場合、関数は true を返し、そうでなければ false を返すものとします。

2.2 メイン コマンドの Execute() 関数から、isRightTemplate() 関数を呼び出します。ドキュメントが正しいテンプレートでない場合、コマンドは実行を停止します:

```
<C#>
public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements )
{
    // objects for the top level access
    //
    _rvtApp = commandData.Application.Application;
    _rvtDoc = commandData.Application.ActiveUIDocument.Document;

    // (0) This command works in the context of family editor only.
    //      We also check if the template is for an appropriate category if
needed.
    //      Here we use a Column(i.e., Metric Column.rft) template.
    //      Although there is no specific checking about metric or imperial,
our lab only works in metric for now.
    //
    if( !isRightTemplate( BuiltInCategory.OST_Columns ) )
    {
        Util.ErrorMessage( "Please open 柱(メートル単位).rft" );
        return Result.Failed;
    }

    ...
}
</C#>
```

注意:メッセージボックスの表示を単純化するために、ヘルパー クラス Util を使用しています。付録 B にてコード サンプルを参照できます。

3. 押し出しで単純なソリッドを作成する

テンプレートが有効だと判断できたら、単純なソリッドの作成を開始します。ここで、長方形のプロファイルを定義して、与えられた高さでそれを押し出します。

3.1 クラスに次の関数を加えてください。このコードは、単純な矩形形状を持つプロファイルを定義します:

```
<C#>
// =====
//      (1.1) create a simple rectangular profile
```

```

// =====
CurveArrArray createProfileRectangle()
{
    //
    // define a simple rectangular profile
    //
    //      3      2
    //      +---+
    //      |    | d      h = height
    //      +---+
    //      0      1
    //      4      w
    //

    // sizes (hard coded for simplicity)
    // note: these need to match reference plane. otherwise, alignment
won't work.
    // as an exercise, try changing those values and see how it behaves.
    //
    double w = mmToFeet(600.0);
    double d = mmToFeet(600.0);

    // define vertices
    //
    const int nVerts = 4; // the number of vertices

    XYZ[] pts = new XYZ[] {
        new XYZ(-w / 2.0, -d / 2.0, 0.0),
        new XYZ(w / 2.0, -d / 2.0, 0.0),
        new XYZ(w / 2.0, d / 2.0, 0.0),
        new XYZ(-w / 2.0, d / 2.0, 0.0),
        new XYZ(-w / 2.0, -d / 2.0, 0.0) };

    // define a loop. define individual edges and put them in a curveArray
    //
    CurveArray pLoop = _rvtApp.Create.NewCurveArray();
    for( int i = 0; i < nVerts; ++i )
    {
        Line line = Line.CreateBound( pts[i], pts[i + 1] );
        pLoop.Append( line );
    }

    // then, put the loop in the curveArrArray as a profile
    //
    CurveArrArray pProfile = _rvtApp.Create.NewCurveArrArray();
    pProfile.Append( pLoop );
    // if we come here, we have a profile now.

    return pProfile;
}
</c#>

```

この関数内で使用するヘルパー関数が一つがあります。

mmToFeet()

Revit はアプリケーション内部では単位にフィートを使用します。API によって寸法を定義する場合は、常に、単位を変換する必要があります。mmToFeet() のコードは、このドキュメントの終りにあるセクション 付録 A に記述されています。このクラスの終わりに コピー&ペースト してください。

この実習では、API の学習を目的とし、コードの判読性を優先するために、長方形の頂点と実際の寸法をハードコーディングしています。サイズは、実際には柱ファミリ テンプレートの中で事前に定義される参照面間の距離から取得します。異なるテンプレートを使用している場合は、それらの値を調節する必要があるでしょう。プロファイルは CurveArrArray(あるいはカーブ配列のコレクション)として定義されます。

テンプレートを開いて、下記の項目を確認してみましょう:

- テンプレート内で使用される寸法
- テンプレートに事前に用意された参照面の名前
- 平面図に正面図での見た目(表現)

実習の全体を通して、それらを参照することに注意してください。

3.2 上記で定義したプロファイルを使用して、押し出しからソリッドを作成します。下記の関数をクラス コードに加えてください:

```
<C#>
// =====
// (1) create a simple solid by extrusion
// =====
Extrusion createSolid()
{
    //
    // (1) define a simple rectangular profile
    //
    //      3      2
    //    +---+
    //    |   | d    h = height
    //    +---+
    //    0      1
    //    4    w
    //
    CurveArrArray pProfile = createProfileRectangle();
    //
    // (2) create a sketch plane
    //
    // we need to know the template. If you look at the template (Metric
    Column.rft) and "Front" view,
```



```

        // you will see "Reference Plane" at "Lower Ref. Level". We are going
to create an extrusion there.
        // findElement() is a helper function that find an element of the given
type and name. see below.
        //
        ReferencePlane pRefPlane = findElement( typeof( ReferencePlane ), "参照
面" ) as ReferencePlane;
        SketchPlane pSketchPlane = SketchPlane.Create(_rvtDoc, pRefPlane.
GetPlane());

        // (3) height of the extrusion
        //
        // once again, you will need to know your template. unlike UI, the
alignment will not adjust the geometry.
        // You will need to have the exact location in order to set alignment.
        // Here we hard code for simplicity. 4000 is the distance between Lower
and Upper Ref. Level.
        // as an exercise, try changing those values and see how it behaves.
        //
        double dHeight = mmToFeet( 4000.0 );

        // (4) create an extrusion here. at this point. just an box, nothing
else.
        //
        bool bIsSolid = true;
        return _rvtDoc.FamilyCreate.NewExtrusion( bIsSolid, pProfile,
pSketchPlane, dHeight );
    }
</c#>

```

この関数の中では、新たに2つめのヘルパー関数が使用されています。

findElement()

このヘルパー関数は、引数に与えられたタイプと名前から要素を検索します。例えば、参照面、レベルやビューを見つけるために使用することが出来ます。完全なコードは、このドキュメントの終わりに記述されています(付録A)。このクラスの終わりにコピー&ペーストしてください。

上記コードの一番下で、以下のメソッドを呼び出しています。

_rvtDoc.FamilyCreate.NewExtrusion()。

これは、押し出しを定義するために使用する主要なメソッドです。引数として、Solid/Void フラグ、プロファイル、スケッチ平面と高さを使用します。スケッチ平面を定義するためには、定義済みの参照面のうちの1つを使用します(例、上記コードやテンプレート内の"参照面")。

ここでは、再度、高さ情報をハードコーディングしています。この値はテンプレート由来で、上部と下部の参照レベル間の距離です(例えば、テンプレートの正面図の中でこれを確認してみてください)。

3.3 メインのコマンド機能から上記の関数を呼び出してください。createSolid () 関数は、押し出しタイプのオブジェクトを返します:

```
<C#>
public IExternalCommand.Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements )
{
    ...

    // (0) This command works in the context of family editor only.
    ...

    if ( !isRightTemplate( BuiltInCategory.OST_Columns ) )
    {
        Util.ErrorMsg( "Please open Metric Column.rft" );
        return IExternalCommand.Result.Failed;
    }

    // (1) create a simple extrusion. just a simple box for now.
    Extrusion pSolid = createSolid();

    // We need to regenerate so that we can build on this new geometry
    _rvtDoc.Regenerate();

    ...
}
</C#>
```

3.4.ソリッドが正確に作成されるかどうかテストするため、この時点でコードをビルドし、実行する準備が整いました。どのようにソリッドが作成されるか試してみてください。

下記のようにアドイン マニフェスト ファイルを作成して、それを Revit のアドインディレクトリに配置します。もちろん、お使いの環境にそれぞれのパラメータを調節する必要があります。ここで、VisibilityMode が「 NotVisibleInProject 」に設定されている点に注意してください。これは、実装したコマンドが、Revit プロジェクトとしてではなく、ファミリ エディタ モードで動作することを意味します。

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>FamilyCs.dll</Assembly>
    <AddInId>98F162FC-90E3-411e-BEC8-75D403BBF11A</AddInId>
    <FullClassName>FamilyCs.RvtCmd_FamilyCreateColumnRectangle</FullClassName>
  </AddIn>
  <Text>Family Labs Create Rectangular Column</Text>
  <Description>Family API lab 1 to create rectangular column</Description>
</RevitAddIns>
```

```

    <VisibilityMode>NotVisibleInProject</VisibilityMode>
    <AccessibilityClassName>Revit.Samples.SampleAccessibilityCheck </Accessib
ilityClassName>
    <VendorId>ADNP</VendorId>
    <VendorDescription>Autodesk, Inc. www.autodesk.com</VendorDescription>
  </AddIn>
</RevitAddIns>

```

「柱(メートル単位).rft」テンプレートを使用して、ファミリ エディタを起動してください。

コマンドを実行した後で、柱が意図したように動作しないかもしれません。これは、この段階では予想通りの振る舞いです。ファミリはパラメトリック のオブジェクトです。上記で定義したソリッドは、モデルの初期状態です。これから、この柱ファミリにパラメトリックな振る舞いをさせるように機能を追加していきます。

4. 位置合わせの追加

次のステップは、ソリッドの各面に対応する参照面との間に位置合わせ拘束を追加します。これは、柱をパラメトリック方式で動作させるために必要です。これにより、ユーザが柱の寸法を変更すると、サイズが自動調節される機能が追加されます。

4.1 次の関数をクラスに加えてください。この関数は、6つの位置合わせの処理を追加します(ボックス形状の6つの各面に):

```

<C#>
// =====
//   (2) add alignments
// =====
void addAlignments( Extrusion pBox )
{
    //
    // (1) we want to constrain the upper face of the column to the "Upper
Ref Level"
    //

    // which direction are we looking at?
    //
    View pView = findElement( typeof( View ), "正面" ) as View;

    // find the upper ref level
    // findElement() is a helper function. see below.
    //
    Level upperLevel = findElement( typeof( Level ), "上基準レベル" ) as
Level;
    Reference ref1 = upperLevel.GetPlaneReference();

    // find the face of the box
    // findFace() is a helper function. see below.

```

```

//
PlanarFace upperFace = findFace( pBox, new XYZ( 0.0, 0.0, 1.0 ) ); //
find a face whose normal is z-up.
Reference ref2 = upperFace.Reference;

// create alignments
//
_rvtDoc.FamilyCreate.NewAlignment( pView, ref1, ref2 );

//
// (2) do the same for the lower level
//

// find the lower ref level
// findElement() is a helper function. see below.
//
Level lowerLevel = findElement( typeof( Level ), "下基準レベル" ) as
Level;
Reference ref3 = lowerLevel.GetPlaneReference();

// find the face of the box
// findFace() is a helper function. see below.
PlanarFace lowerFace = findFace( pBox, new XYZ(0.0, 0.0, -1.0) ); //
find a face whose normal is z-down.
Reference ref4 = lowerFace.Reference;

// create alignments
//
_rvtDoc.FamilyCreate.NewAlignment( pView, ref3, ref4 );

//
// (3) same idea for the Right/Left/Front/Back
//
// get the plan view
// note: same name maybe used for different view types. either one
should work.
View pViewPlan = findElement( typeof( ViewPlan ), "下基準レベル" ) as
View;

// find reference planes
ReferencePlane refRight = findElement( typeof( ReferencePlane ), "右" )
as ReferencePlane;
ReferencePlane refLeft = findElement( typeof( ReferencePlane ), "左" )
as ReferencePlane;
ReferencePlane refFront = findElement( typeof( ReferencePlane ), "正面
" ) as ReferencePlane;
ReferencePlane refBack = findElement( typeof( ReferencePlane ), "背面" )
as ReferencePlane;

// find the face of the box
PlanarFace faceRight = findFace( pBox, new XYZ( 1.0, 0.0, 0.0 ) );
PlanarFace faceLeft = findFace( pBox, new XYZ( -1.0, 0.0, 0.0 ) );
PlanarFace faceFront = findFace( pBox, new XYZ( 0.0, -1.0, 0.0 ) );

```

```

PlanarFace faceBack = findFace( pBox, new XYZ( 0.0, 1.0, 0.0 ) );

// create alignments
//
_rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refRight.GetReference(),
faceRight.Reference );
_rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refLeft.GetReference(),
faceLeft.Reference );
_rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refFront.GetReference(),
faceFront.Reference );
_rvtDoc.FamilyCreate.NewAlignment( pViewPlan, refBack.GetReference(),
faceBack.Reference );
}
</c#>

```

この関数内では、新たに 3 つめのヘルパー関数を使用されています。

findFace()

このヘルパー関数は、押し出しソリッドから与えられた法線で平面を見つけます。完全なコードは、このドキュメントの終わりに記載されています(付録 A)。このクラスの終わりに コピー&ペースト してください。

ソリッドの上部の面と参照面「上参照レベル」の間に位置合わせを追加するところで、コードの最初の部分に注目してください。ここで1つ理解できれば、コードの残り部分でも同じように理解できるはずです。

```
_rvtDoc.FamilyCreate.NewAlignment(pView, ref1, ref2)
```

これは、新しい位置合わせを作成する主要なメソッドです。引数としてビューと2つの参照を渡します。UIで行うように、上部の面を上参照面への位置合わせするために、側面からモデルを見ています。ここでは、正面ビューを使用します。findElement() と findFace() はヘルパー関数です。

注意すべきなのは、UI と異なり、API メソッドである NewAlignment() は、自動的に計算してモデルのジオメトリを調節しないという点です。下記は、RevitAPI.chm ファイルからの抜粋です：

“These references must be already geometrically aligned (this function will not force them to become aligned).”

NewAlignment() を呼に出す前に、それらが同じ位置にあることを確かめる必要があります。

4.2 メイン コマンドの Execute() から、addAlignments(pSolid) を呼び出します：

```

<c#>
public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements )
{
    ...

    // (1) create a simple extrusion. just a simple box for now.

```

```

Extrusion pSolid = createSolid();

// (2) add alignment
addAlignments( pSolid );
...

```

</C#>

4.3 この時点でコードをビルドし、実行する準備が整いました。この柱は、参照を変更した際に正しく応答するはずです(例えば、レベルの高さなど)。

5. タイプを追加する

「幅」と「奥行」の寸法を変えたタイプを 2 つ追加します:

- 600 x 900
- 1000x300
- 600 x 600

5.1 次の関数をクラスに加えてください:

```

<C#>
// =====
// (3) add types
// =====
void addTypes()
{
    // addType(name, Width, Depth)
    //
    addType( "600x900", 600.0, 900.0 );
    addType( "1000x300", 1000.0, 300.0 );
    addType( "600x600", 600.0, 600.0 );
}

// add one type
void addType( string name, double w, double d )
{
    // get the family manager from the current doc
    FamilyManager pFamilyMgr = _rvtDoc.FamilyManager;

    // add new types with the given name
    //
    FamilyType type1 = pFamilyMgr.NewType( name );

    // look for 'Width' and 'Depth' parameters and set them to the given
value
    //
    // first 'Width'
    //
    FamilyParameter paramW = pFamilyMgr.get_Parameter( "幅" );

```

```

double valW = mmToFeet( w );
if ( paramW != null )
{
    pFamilyMgr.Set( paramW, valW );
}

// same idea for 'Depth'
//
FamilyParameter paramD = pFamilyMgr.get_Parameter( "奥行" );
double valD = mmToFeet( d );
if ( paramD != null )
{
    pFamilyMgr.Set( paramD, valD );
}
}
</c#>

```

ここで覚えておく必要のあるクラスは、ドキュメントのファミリ マネージャ オブジェクトです。このオブジェクトは次のようにアクセスすることができます:

```
_rvtDoc.FamilyManager
```

一度、ファミリマネージャを取得したら、NewType メソッドを使用して、新しいタイプを作成することができます:

```
pFamilyMgr.NewType(name)
```

次のメンバ関数を使用して、当該パラメータにアクセスすることができます:

```
pFamilyMgr.Parameter( "幅" )
```

その後で、次のように値を設定します:

```
pFamilyMgr.Set(paramW, valW)
```

上記のコードは3つのタイプを定義します。

5.2 メイン コマンドの Execute() から、addTypes() を呼び出します:

```

<c#>
public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements )
{
    ...
    // (2) add alignment
    addAlignments(pSolid)

    // (3) add types
    addTypes();
}

```

```

        // finally, return
        return Result.Succeeded;
    }
</c#>

```

5.3 コードをビルドし、実行してみましょう。

6. 作成した柱ファミリをテストする

柱ファミリが正しく作成されていれば、この時点でコードをビルドし、実行することができます。

「柱(メートル単位).rft」テンプレートを使用して、ファミリ エディタを起動してください。

コマンドを実行した後に、タイプ ダイアログで3つのタイプが作成されるかどうかチェックして、それを適用して柱がサイズに従って変更されるか確かめてください。

次の実習では、柱のプロファイルを修正して、参照面、パラメータと寸法を追加する方法を学習します。

付録 A. Lab1 の中で使われるヘルパー関数

Lab1 では、次のヘルパー関数を使用します。必要に応じて下記のコードをコピー&ペーストで作成中のコードに貼り付けてください。

- findFace() — 与えられた押し出しソリッドから、与えられた法線で平面を見つける
- findElement() — 与えられたタイプおよび名前の要素を見つける(例、参照やレベルの検索など)
- mmToFeet() — 単位をミリメートルからフィートへ変換する

```

<C#>
#region Helper Functions

// =====
//  helper function: find a planar face with the given normal
//  =====
PlanarFace findFace(Extrusion pBox, XYZ normal)
{
    // get the geometry object of the given element
    //
    Options op = new Options();
    op.ComputeReferences = true;
    IEnumerable<GeometryObject> geomObjs =
pBox.get_Geometry(op).AsEnumerable();

    // loop through the array and find a face with the given normal
    //

```



```

foreach( GeometryObject geomObj in geomObjs )
{
    if( geomObj is Solid ) // solid is what we are interested in.
    {
        Solid pSolid = geomObj as Solid;
        FaceArray faces = pSolid.Faces;
        foreach( Face pFace in faces )
        {
            PlanarFace pPlanarFace = pFace as PlanarFace;
            if( (pPlanarFace != null) && pPlanarFace.
FaceNormal.IsAlmostEqualTo( normal ) ) // we found the face
            {
                return pPlanarFace;
            }
        }
    }

    // will come back later as needed.
    //
    //else if (geomObj is Instance)
    //{
    //}
    //else if (geomObj is Curve)
    //{
    //}
    //else if (geomObj is Mesh)
    //{
    //}
}

// if we come here, we did not find any.
return null;
}

// =====
//  helper function: find an element of the given type and the name.
//  You can use this, for example, to find Reference or Level with the
given name.
// =====
Element findElement(Type targetType, string targetName)
{
    // get the elements of the given type
    //
    FilteredElementCollector collector = new
FilteredElementCollector(_rvtDoc);
    collector.WherePasses(new ElementClassFilter(targetType));

    // parse the collection for the given name
    // using LINQ query here.
    //
    var targetElems = from element in collector where
element.Name.Equals(targetName) select element;
    List<Element> elems = targetElems.ToList<Element>();
}

```

```

        if ( elems.Count > 0 ) { // we should have only one with the given
name.
            return elems[0];
        }

        // cannot find it.
        return null;    }

// =====
//  helper function: convert millimeter to feet
// =====
double mmToFeet( double mmVal )
{
    return mmVal / 304.8;
}

#endregion // Helper Functions
</c#>

```

付録 B. C# のメッセージボックスを表示するヘルパー関数

実習では、C#の中のメッセージボックスの表示を簡略化するために、次のヘルパー関数を使用します。新しい.cs ファイルを追加して、必要に応じて下記のコードをコピー＆ペーストで作成中のコードに貼り付けてください。

```

<c#>
#region Namespaces
using System;
using System.Diagnostics;
using WinForms = System.Windows.Forms;
#endregion // Namespaces

namespace FamilyLabsCS
{
    public class Util
    {
        #region Formatting and message handlers
        public const string Caption = "Revit Family API Labs";

        /// <summary>
        ///  MessageBox wrapper for informational message.
        /// </summary>
        public static void InfoMsg( string msg )
        {
            Debug.WriteLine( msg );
            WinForms.MessageBox.Show( msg, Caption, WinForms.MessageBoxButtons.OK,
WinForms.MessageBoxIcon.Information );
        }
    }
}

```

```
/// <summary>
/// MessageBox wrapper for error message.
/// </summary>
public static void ErrorMsg( string msg )
{
    WinForms.MessageBox.Show( msg, Caption, WinForms.MessageBoxButtons.OK,
WinForms.MessageBoxIcon.Error );
}
#endregion // Formatting and message handlers
}
}
</c#>
```