

eTlipse

// uso da // programação computacional // na metodologia BIM



joel diniz
edson andrade

Versão PTB 20200731 - eBook gratuito

Macro - SharpDevelop / Programação Visual - Dynamo / API do Revit -
C# com Visual Studio

JOEL DINIZ



Mestre em Processos Construtivos. Engenheiro Civil. Bacharel em Ciência da Computação. MBA em BIM. Pós-Graduado em Georreferenciamento e Estruturas de CA.

EDSON ANDRADE



Arquiteto e Urbanista. MBA em Plataforma BIM.

Apresentação

Esta obra demonstra o uso da programação na Metodologia BIM, especificamente com Revit (fazendo uso de sua API, *Application Programming Interface*), Dynamo (como exemplo de linguagem visual) e C# (como exemplo de linguagem “textual”).

Importante ressaltar que esta edição se destina àqueles que ainda são iniciantes, ou pretendem iniciar os estudos de programação computacional para a metodologia BIM. Alguns tópicos básicos ainda não estão desenvolvidos, mas registram o conhecimento que colabora com o entendimento da programação, conhecimento que recomendamos ser estudado, e que oportunamente iremos desenvolver neste trabalho, ainda que suscintamente.

Este material está sendo desenvolvido para futuramente ser lançado como eBook, ele é fruto de uma extensa pesquisa, e da formação nas áreas de Engenharia Civil, Arquitetura, Ciências da Computação e BIM dos autores. Utilizamos diversos exemplos que foram disponibilizados na Internet de forma aberta por outros autores. No final registramos referências que pode ser útil para pesquisa direta de fontes importantes. Uma vez que não se reinventa a roda na programação, os recursos fazem uso de bibliotecas de classes com inúmeras utilidades. É importante atentar para as versões das bibliotecas utilizadas, infelizmente, por questão de compatibilidade com as partes integradas nem sempre é possível o uso das bibliotecas mais atuais.

Uma vez que os exemplos resultam de pesquisas e experiências anteriores, e ainda em virtude da linguagem de programação utilizada, estes podem conter trechos em inglês, mas que não prejudica o entendimento. Em alguns trechos tentamos usar apenas os comentários em português para ajudar no entendimento, isto porque nos propomos a disponibilizar um material que contenha o máximo em português, mas de qualquer forma reconhecemos a importância do inglês e mesmo encorajamos seu estudo. Evitamos pontuação nos comentários dos códigos, o que pode parecer erro ortográfico em alguns trechos. Os exemplos contêm alguns comentários adicionais que servem apenas como dicas, ou para marcar possibilidade alternativa. Tentamos evitar exemplos parciais para facilitar a replicação dos mesmos, incluindo até as diretrizes que se repetem nos exemplos.

Esperamos que este material facilite o ingresso nesta área ainda carente de fontes, fato este que nos motivou a criar este material. Infelizmente a ausência de mais fontes e mesmo a falta de disponibilidade de alguns profissionais que já dominam o assunto em tirar dúvidas, representam uma enorme barreira, empecilhos estes que motivou o desenvolvimento deste material para que outros não tenham tanta dificuldade como enfrentemos. Aproveite e divulgue o material!

A versão mais atualizada deste eBook está disponível na área de download da eTlipse, portanto, antes de usar, certifique-se que está com a versão mais atualizada visitando o site:

<https://www.etlipse.com/>

A large, light-grey watermark of the "eTlipse" logo is positioned diagonally across the page, starting from the bottom-left and ending near the top-right.

Notas da versão:

- ✓ Versão: nº PTB 20200731

Atualizações:

- ✓ Revisão do texto da versão anterior segundo feedback. Especial colaboração nesta versão revisada por Guilherme Almeida.
- ✓ Disponibilizado mais exemplos da API do Revit.
- ✓ Desenvolvimento dos conceitos básicos de computação e programação.
- ✓ eTLipse Template para WPF com Revit API.
- ✓ Trabalho com visões e caixas de corte no Revit.
- ✓ Implementação simples de uma aplicação usando MVVM, Model View View Model, com WPF.

Futuras Melhorias:

- ✓ Desenvolver conceitos básicos da programação e computação.
- ✓ Constante correção ortográfica, algo que requer especial atenção, visto que a grande quantidade de código atrapalha a correção automática.
- ✓ Novos exemplos práticos.

Tópicos Abordados

Conceitos Básicos	10
Estrutura do Computador.....	10
Linguagens de programação.....	11
Algoritmos.....	12
Estruturas de dados.....	12
Linguagem C# - Programação "Textual".....	13
Variáveis.....	13
Operadores.....	14
Estruturas de programação na linguagem C#.....	15
Estrutura Condicional - if .. else (Tomada de decisão).....	15
Laço - for	15
Laço - foreach	16
Laço - while	16
Laço - do ... while.....	16
Exceções - try ... catch ... finally.....	16
Estrutura de classe	17
Macro - C# no SharpDevelop.....	18
Exemplo - Prático Tipos Paredes	18
Programação Visual - Dynamo.....	28
Exemplo Prático Transversinas.....	31
Exemplo - Codificação para BIM 4D	38
Exemplo - Cerca Viva	39
Uso de Código no Dynamo	40
Linguagem Base.....	40
Geometria Básica	42

Geometrias Primitivas	46
Matemática de Vetor	49
Expressões de Intervalo	52
Coleções	56
Funções	58
Matemática	61
Curvas: Pontos Interpretados e de Controle	64
Translação, Rotação e Outras Transformações	67
Condicionais e Lógica Booleana	70
Looping	73
Guias de Replicação	75
Coleções Classificadas e Coleções Irregulares	77
Geração de Elementos com Python	79
Superfícies: Interpretado, Pontos de Controle, Loft, Revolve	85
Criação de Módulo Dynamo no Visual Studio com C#	98
Exemplo - Ola Dynamo	104
Exemplo - Média	105
Exemplo - Saída de Vários Valores (Separa Par/ímpar)	106
Exemplo - Geometria	108
Exemplo - Elementos do Revit	110
Exemplo - Parede desenha texto	114
Exemplo - Usando WPF com Nodes Dynamo	119
Publicando pacotes no Dynamo para o Package Manager	134
Visual Studio	136
WPF - Windows Presentation Foundation	136
Exemplo - Projeto WPF para Gerar CSV	137

Utilizando o Excel como Base de Dados.....	143
Exemplo - Interagindo com o Excel.....	143
Utilizando arquivos XML como Base de Dados	149
Exemplo - Interagindo com Arquivo XML.....	149
UML - Unified Modeling Language.....	159
Utilizando padrão MVVM com WPF.....	160
Revit API.....	173
Exemplo - Projeto "Olá Revit!".....	173
Elaboração de Manifesto que Habilita o Plugin.....	181
Executando o plug-in.....	183
Personalizar Ribbon Painel.....	184
Revit API com WPF - Template eTlipse.....	187
Exemplos Iniciais Utilizando a API do Revit.....	192
Exemplo - Coletando Informação de Elementos Selecionados.....	192
Exemplo - Revit com WPF	193
Exemplo - Adoção de <i>Using</i> e <i>Transaction</i>	200
Exemplo - Determinando Valor de Parâmetro de Objeto Selecionado.....	201
Exemplo - Obtendo Valor de Parâmetro de Objeto Selecionado	202
Exemplo - Exportando para Excel.....	204
Uso de Filtros e Técnicas para Seleção de Elementos.....	206
Exemplo - Filtrando Elementos	206
Exemplo - Usando Filtro de Parâmetro para Selecionar Room pela Área.....	207
Exemplo - Combinando Filtros para Selecionar Portas Instanciadas.....	208
Exemplo - Usando o Padrão Iterator	210
Exemplo - Usando LINQ	211
Exemplo - Criando um Novo Grupo a Partir de uma Seleção	212

Exemplo - Seleção com Interação do Usuário.....	216
Exemplo - Coleções e Interações.....	218
Interagindo com parâmetros.....	219
Exemplo - Obtendo Parâmetros de Dado Elemento.....	219
Exemplo - Obtendo Parâmetros Baseado no Tipo Definido.....	221
Exemplo - Obtendo Parâmetros Baseado em BuiltInParameter	222
Exemplo - StoreType e Conversão para Unidade Interna.....	223
Exemplo - AsValueString e SetValueString.....	225
Exemplo - Parâmetros Relacionados.....	226
Interagindo com Elementos.....	228
Exemplo - Movendo Elementos.....	228
Exemplo - Rotacionando Elementos.....	231
Exemplo - Espelhando Elementos.....	234
Exemplo - Deletando Elementos e Agrupando Elementos	235
Interagindo com Vistas.....	238
Exemplo - Elementos em Vista Ativa.....	238
Exemplo - Tipos de Vista.....	239
Exemplo - Criando Vistas Perspectiva e Isométrica 3D.....	242
Interagindo com Caixas de Corte.....	245
Exemplo - Caixa de Corte.....	245
Referências.....	249

Conceitos Básicos

Inicialmente apresentamos princípios da computação, que servem apenas como breve revisão, ou fonte para incentivar uma pesquisa apropriada para aqueles que desejam se aprofundar. O importante dos conceitos iniciais é que estes permitem explorar as bases necessárias para o entendimento do funcionamento da programação.

Estrutura do Computador

O processador representa a principal unidade de processamento do computador, onde estão gravadas as instruções básicas de processamento. É, portanto, importante entender que a lógica computacional adotada, o algoritmo elaborado para a solução do problema, será definido como um conjunto de soluções menores que possam ser processadas. A ideia de dividir um problema complexo em uma série de problemas menores, mais fáceis de resolver, é a estratégia adota pela computação. Uma vez que estamos falando de um recurso de custo relativamente elevado, as diferentes técnicas para processamento das soluções menores levam a arquiteturas que definem os diferentes tipos de processadores comercializados.

A memória consiste em cada célula onde é alocada a informação, dado, a ser processado. Seu tamanho e velocidade pode ser relacionado com seu custo:

Rápida – Custo elevado;

Grande capacidade – Custo reduzido.

Os dispositivos de entrada de dados permitem a interação de forma que o computador seja alimentado com dados para processamento. Exemplo podemos citar teclado e mouse.

Os dispositivos de saída de dados servem para exibir o resultado dos dados processados. Exemplo podemos citar vídeo e impressora.

Linguagens de programação

A programação representa a definição de um conjunto de instruções a serem executadas pelo computador, tornando-se necessário o entendimento destas instruções. Para o efetivo entendimento utilizamos linguagens específicas para as quais o computador está preparado para interpretar.

Podemos classificar as linguagens segundo certos níveis, conforme sua “proximidade” com as instruções base do processador, que trabalha em uma organização de estrutura binária, nos computadores convencionais. De forma mais genérica teremos:

Linguagem de baixo nível – Assembly;

Linguagem de alto nível.

Máquina Virtual:

Java;

.Net;

Linguagens:

Procedural – Fortran, Basic;

Estruturada – Pascal, C;

Orientada a Objetos - C++, C#, Java, Python, Ruby;

Computação Distribuída – Ada;

Programação Visual – Dynamo, Grasshopper.

Outros Exemplos: JavaScript, Visual Basic .NET, PHP, MATLAB, Perl, Ruby, Delphi / Object Pascal, SQL, Visual Basic ...

Algoritmos

Estruturas de dados

Estruturas usuais:

Condições;

Repetições;

Atribuição.

eTlipse

Linguagem C# - Programação "Textual"

Uma poderosa linguagem de programação, não apenas para BIM, mas para todo tipo de aplicativo, é a linguagem CSharp ou C#.

Variáveis

Os dados que precisamos operar na programação podem, ou mesmo devem, estar armazenados em memória de grande capacidade, visto a necessidade de um grande volume, precisam usar uma memória de menor custo, em contrapartida, uma memória de acesso mais lento. Obviamente é desejado que nossos programas rodem com enorme velocidade, esta necessidade nos leva ao uso de memória de rápido acesso, e nelas precisamos armazenar os dados que estejam mais acessíveis. Estes dados para uso dos programas são armazenados segundo uma estrutura de dados, que define seu tipo, e recebem um determinado nome para identificação, a este recurso denominamos variáveis. A ideia do nome está relacionada ao que usamos na matemática, e por ser, embora tenhamos situações específicas, um valor que podem ser modificados pelo programa.

Algumas boas práticas são importantes atentar para o nome da variável, tais como usar nomes lógicos, segundo o que irá representar, iniciar com letras minúsculas, para diferenciar das classes por exemplo, assim como evitar todas as letras maiúsculas. Fundamental lembrar que outros, mesmo você após longo período, precisarão ler o código, para possíveis manutenções por exemplo, e ter um código legível é fundamental.

Existem duas abordagens bastante usadas para nomear variáveis, considerando que não raro teremos um nome composto por mais de uma palavra unidas sem nenhum separador. Na convenção PascalCasing todas as palavras iniciam com a primeira letra em maiúsculo, neste caso é mais recomendado que seja usado para nome de classes, métodos, propriedades, enumeradores, interfaces, constantes, campos somente leitura e namespaces. A outra abordagem, mais adequada para variáveis que representam instâncias, é o camelCasing, onde cada palavra inicia com maiúscula, exceto a primeira palavra que deve iniciar com minúscula.

Um cuidado importante é que cada variável possuiu nome único, e que existem palavras reservadas pela linguagem que não podem ser usadas como variáveis. Tais como abstract, event, new, base, extern, object, bool, false, operator, break, finally, out, byte, fixed, override.

Inteiros (Byte, Short, Integer, Long) – 1 a 8 bytes.

Decimais (Float, Double) – Valores com ponto flutuante.

String (Char, String) – Caracteres Unicode; 2 bilhões de caracteres.

Booleano (Boolean) – Verdadeiro ou falso (True, False).

Outros: Data, Vetores, Constantes, Operadores, Enumerações.

Operadores

Comparação:

> maior que;

< menor que;

== igual a;

!= diferente de;

>= maior ou igual a;

<= menor ou igual a;

“like” strings.

Like:

* n caracteres adicionais;

? um caractere;

dígito {0-9};

intervalos [0-9] [a-z] ...

Lógicos:

& AND (E);

|| OR (OU);

XOR;

NOT.

Estruturas de programação na linguagem C#

Estrutura Condicional - if .. else (Tomada de decisão)

```
int x = 5; int y = 10; int a;
```

```
if (x == y) {
```

```
    a = 10;
```

```
} else if (x <= y) {
```

```
    a = x + y;
```

```
} else {
```

```
    a = 5;
```

```
}
```

Laço - for

```
int length = 10;
```

```
for (int i = 0; i < length; i++) {
```

```
    a = x + i;
```

```
}
```

Laço - foreach

```
string[] collection = new string[] { "Edson", "Rogerio", "Joel" };

foreach (var item in collection) {

    Console.WriteLine(item.ToString());

}
```

Laço - while

```
while (y < 100) {

    y++;

}
```

Laço - do ... while

```
do {

    y++;

} while (y < 100);
```

Exceções - try ... catch ... finally

```
public class eOutOfRange : Exception { }

public MainWindow() { InitializeComponent();

    int x = 5; y = 10; int a;

    try {

        a = x + y; if(x < y) { throw new eOutOfRange();} // catch(eOutOfRange){/**/}

    } catch (Exception e) {
```

```
Console.WriteLine(e.Message);

} finally {

    Console.WriteLine("Bye!");

}
```

Estrutura de classe

```
using System; using System.Collections.Generic; using System.Linq; using System.Text; using  
System.Threading.Tasks;
```

```
namespace WpfAppDynamo{

    // Classe

    public class NewClasse{

        // Atributos

        private int a; public int b;

        // Método

        public int Action() {

            a = a + b; return a;

        }

    }

}
```

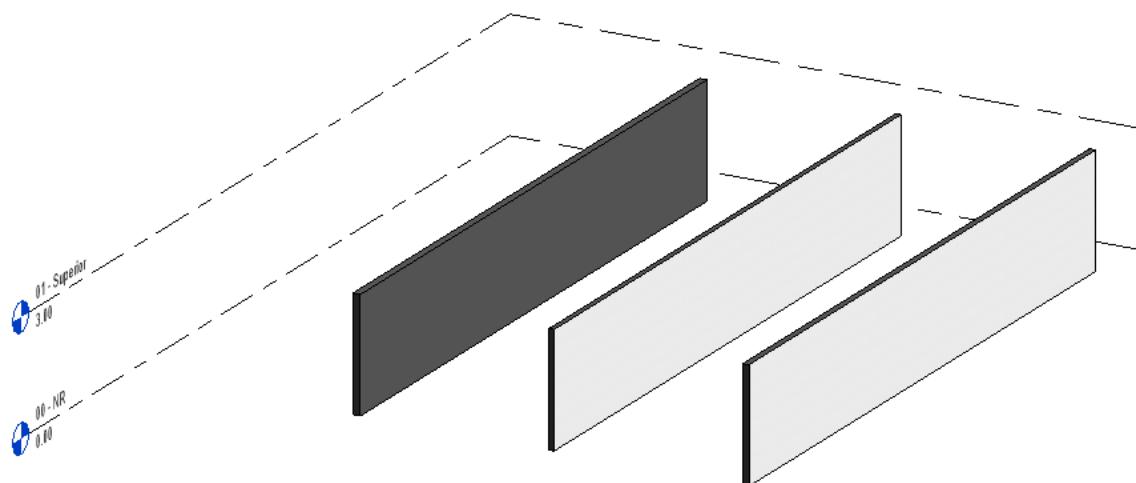
Macro - C# no SharpDevelop

É possível determinar macros no Revit através de linguagem de programação “textual”. Essa é uma forma de usar o editor de programação nativo do Revit, o SharpDevelop.

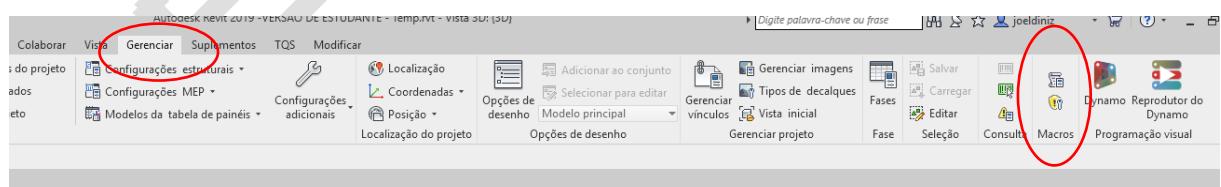
Exemplo - Prático Tipos Paredes

Objetivo deste exemplo é listar os tipos de paredes em uso no modelo ativo, utilizando C# no SharpDevelop.

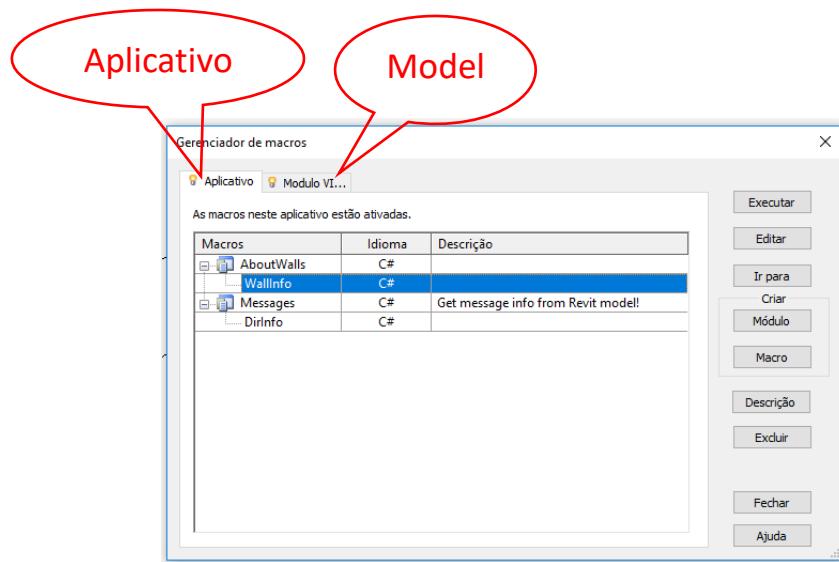
Um modelo precisa ser preparado para a aplicação do exemplo. Criamos 3 paredes definindo cada uma de um tipo diferente.



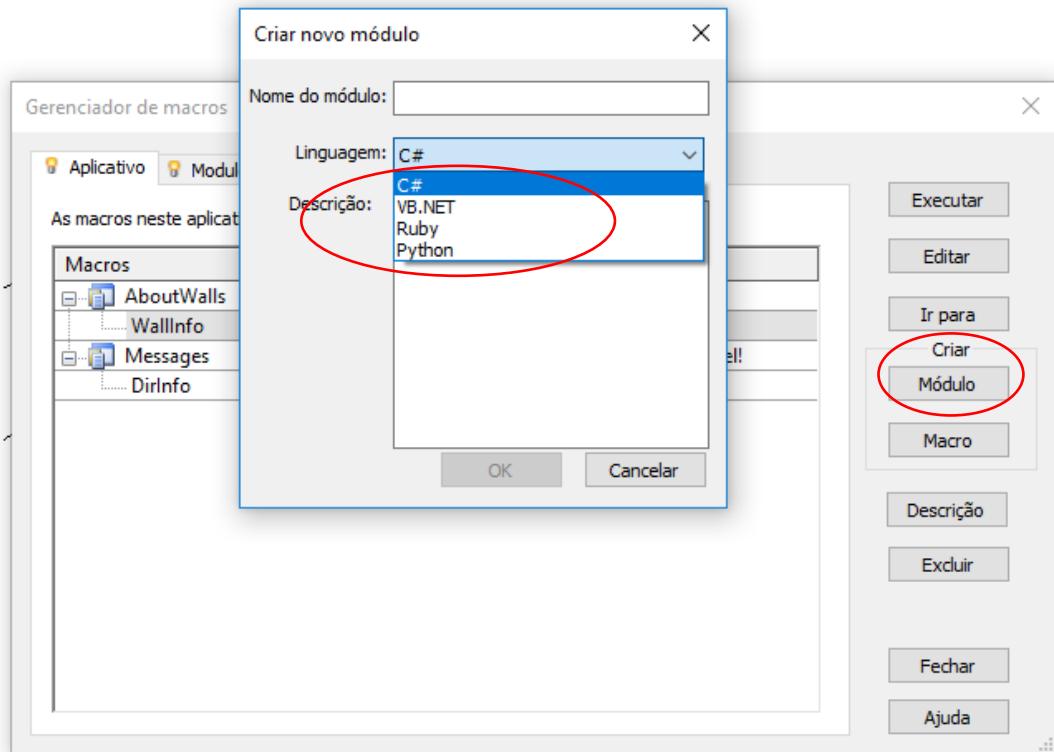
É possível utilizar programação textual no Revit, através da opção de Macros. Esta opção pode ser acessada através da guia **Gerenciar**, opções **Macros**.



O **Gerenciador de macros** é iniciado. Existem duas formas de desenvolver. Na guia aplicativo o macro ficará disponível para qualquer outro modelo que for aberto no computador onde foi desenvolvido. Na guia de módulo, o desenvolvimento ficará disponível apenas para o modelo ativo. Uma advertência de possibilidade de vírus será exibida quando um modelo que contenha macro for executado.



Inicialmente criamos um módulo. Definimos um nome para o módulo. Escolhemos a opção de linguagem que será utilizada. É possível usar C#, VB.NET, Ruby ou Python. A descrição é opcional, mas pode ajudar a compreender a funcionalidade do módulo.



O SharpDevelop é iniciado após a criação do módulo. Esta ação ocorrerá quando novos macros forem criados neste módulo. Para aqueles que já programam, devem notar que o módulo será o namespace e macros serão funções públicas no código.

```

 1  /*
 2   * Created by SharpDevelop.
 3   * User: jrdi
 4   * Date: 05/06/2018
 5   * Time: 22:57
 6   *
 7   * To change this template use Tools | Options | Coding | Edit Standard Headers.
 8   */
 9  using System;
10 using Autodesk.Revit.UI;
11 using Autodesk.Revit.DB;
12 using Autodesk.Revit.UI.Selection;
13 using System.Collections.Generic;
14 using System.Linq;
15
16 namespace Teste
17 {
18     [Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)]
19     [Autodesk.Revit.DB.Macros.AddInId("131232BA-4C3A-49B9-8A65-D1ED3D01EFC")]
20     public partial class ThisApplication
21     {
22         private void Module_Startup(object sender, EventArgs e)
23         {
24         }
25
26         private void Module_Shutdown(object sender, EventArgs e)
27         {
28         }
29
30         Revit Macros generated code
31     }
32 }
33
34
35
36
37
38
39
40

```

Criamos um módulo C# nomeado AboutWalls, ou outro nome que tenha lógica para a funcionalidade a desenvolver. Fechamos o SharpDevelop e verificamos que este aparece listado no gerenciador. Criamos um Macro nomeado WallInfo, ou outro nome lógico relacionado à funcionalidade deste.

Macros	Idioma	Descrição
AboutWalls	C#	
WallInfo	C#	
Messages	C#	Get message info from Revit model!
DirInfo	C#	

```
namespace AboutWalls
{
    [Autodesk.Revit.Attributes.Transaction(Autode
    [Autodesk.Revit.DB.Macros.AddInId("5B05BFD1-3
    public partial class ThisApplication
    {
        private void Module_Startup(object sender
        {

Revit Macros generated code
public void WallInfo(){
    // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
    UIDocument uiDoc = this.ActiveUIDocument;
    Document currentDoc = uiDoc.Document;

    try {
        // Find all walls instances in the document by using c
```

Código completo é exibido na área de trabalho do SharpDevelop. Quando criar o macro a maior parte deste código estará pronto, mas algumas adaptações são úteis. O uso de cores facilita leitura do código. As linhas são numeradas para facilitar a identificação de erros. O código obedece a definição da API do Revit “Application Programming Interface” ou “Interface de Programação de Aplicativos”.

Code editor showing the `AboutWalls.ThisApplication.cs` file:

```

1  /*
2   * Created by SharpDevelop.
3   * User: j3di
4   * Date: 09/05/2018
5   * Time: 00:01
6   *
7   * To change this template use Tools | Options | Coding | Edit Standard Headers.
8   */
9  using System;
10 using Autodesk.Revit.UI;
11 using Autodesk.Revit.DB;
12 using Autodesk.Revit.UI.Selection;
13 using System.Collections.Generic;
14 using System.Linq;
15
16 namespace AboutWalls
17 {
18     [Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)]
19     [Autodesk.Revit.DB.Macros.AddInId("5805BFD1-3515-4F00-9902-E17A8C6D558E")]
20     public partial class ThisApplication{
21         private void Module_Startup(object sender, EventArgs e){
22             }
23
24         private void Module_Shutdown(object sender, EventArgs e){
25             }
26
27         Revit Macros generated code
28         public void WallInfo(){
29             // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
30             UIDocument uiDoc = this.ActiveUIDocument;
31             Document currentDoc = uiDoc.Document;
32
33             try {
34                 // Find all walls instances in the document by using category filter!
35                 ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);
36                 // Apply the filter to the elements in the active document!
37                 // Use shortcut WhereElementIsNotElementType() to find wall instances only
38                 FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
39
40                 IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
41
42                 String prompt = "The walls in the current document are:\n";
43                 foreach (Element e in walls) {
44                     prompt = prompt + e.Name + "\n";
45                 }
46                 TaskDialog.Show("Revit", prompt);
47             } catch (Exception) {
48                 throw;
49             }
50         }
51     }
52 }

```

Existem duas formas de usar comentário no código. Comentários não serão considerados na execução do código. O que estiver entre os caracteres “`/*`” e “`*/`” serão considerados comentários, independente do numero de linhas, conforme figura das linhas 1 a 8. O que estiver após os caracteres “`//`” serão considerados comentários nesta linha apenas, conforme a figura mostra este uso na linha 35.

```

1  /*
2   * Created by SharpDevelop.
3   * User: j3di
4   * Date: 09/05/2018
5   * Time: 00:01
6   *
7   * To change this template use Tools |
8   */

```

```

34:     35:         // Setup Revit UI (uiDoc) and Current
35:         // Document uiDoc = this.ActiveUIDocument
36: 
37:     38:         public void WallInfo()
38:             // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
39:             UIDocument uiDoc = this.ActiveUIDocument;
40:             Document currentDoc = uiDoc.Document;
41: 
42:             try {
43:                 // Find all walls instances in the document by using category filter!
44:                 ElementCategoryFilter filter = new ElementCategoryFilter(BuiltinCategory.OST_Walls);
45:                 // Apply the filter to the elements in the active document!
46:                 // Use shortcut WhereElementIsNotElementType() to find wall instances only
47:                 FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
48: 
49:                 IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
50: 
51:                 String prompt = "The walls in the current document are:\n";
52:                 foreach (Element e in walls) {
53:                     prompt = prompt + e.Name + "\n";
54:                 }
55:                 TaskDialog.Show("Revit", prompt);
56:             } catch (Exception) {
57:                 throw;
58:             }
59:         }
60:     }
61: }

```

A seguir o código da macro, perceba que é uma função pública em C#, nomeada com o nome definido para a macro. O código vai da linha 27 à 56.

```

27:     28:         public void WallInfo()
28:             // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
29:             UIDocument uiDoc = this.ActiveUIDocument;
30:             Document currentDoc = uiDoc.Document;
31: 
32:             try {
33:                 // Find all walls instances in the document by using category filter!
34:                 ElementCategoryFilter filter = new ElementCategoryFilter(BuiltinCategory.OST_Walls);
35:                 // Apply the filter to the elements in the active document!
36:                 // Use shortcut WhereElementIsNotElementType() to find wall instances only
37:                 FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
38: 
39:                 IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
40: 
41:                 String prompt = "The walls in the current document are:\n";
42:                 foreach (Element e in walls) {
43:                     prompt = prompt + e.Name + "\n";
44:                 }
45:                 TaskDialog.Show("Revit", prompt);
46:             } catch (Exception) {
47:                 throw;
48:             }
49:         }
50:     }
51: }

```

Inicialmente é feita a conexão com o Revit e então com o documento, modelo ativo, definindo um objeto Document. Com este objeto é possível interagir com o modelo. Isto é feito nas linhas 36 e 37.

```

27:     28:         public void WallInfo()
28:             // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
29:             UIDocument uiDoc = this.ActiveUIDocument;
30:             Document currentDoc = uiDoc.Document;
31: 
32:             try {
33:                 // Find all walls instances in the document by using category filter!
34:                 ElementCategoryFilter filter = new ElementCategoryFilter(BuiltinCategory.OST_Walls);
35:                 // Apply the filter to the elements in the active document!
36:                 // Use shortcut WhereElementIsNotElementType() to find wall instances only
37:                 FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
38: 
39:                 IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
40: 
41:                 String prompt = "The walls in the current document are:\n";
42:                 foreach (Element e in walls) {
43:                     prompt = prompt + e.Name + "\n";
44:                 }
45:                 TaskDialog.Show("Revit", prompt);
46:             } catch (Exception) {
47:                 throw;
48:             }
49:         }
50:     }
51: }

```

O código é definido em um bloco “try - catch”, para tratamento de exceções, em caso de erro. A exceção pode ser tratada e a mensagem de erro destacada.

```

27 Revit Macros generated code
28 public void WallInfo(){
29     // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
30     UIDocument uiDoc = this.ActiveUIDocument;
31     Document currentDoc = uiDoc.Document;
32
33     try {
34         // Find all walls instances in the document by using category filter!
35         ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);
36         // Apply the filter to the elements in the active document!
37         // Use shortcut WhereElementIsNotElementType() to find wall instances only
38         FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
39
40         IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
41
42         String prompt = "The walls in the current document are:\n";
43         foreach (Element e in walls) {
44             prompt = prompt + e.Name + "\n";
45         }
46         TaskDialog.Show("Revit", prompt);
47     } catch (Exception) {
48         throw;
49     }
50 }

```

Opção de tratamento de erro

```

TaskDialog.Show("Revit", prompt);
} catch (Exception exp) {
    TaskDialog.Show("Failed", exp.ToString());
}

```

Um filtro é definido para agrupar apenas as instâncias de parede.

```

27 Revit Macros generated code
28 public void WallInfo(){
29     // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
30     UIDocument uiDoc = this.ActiveUIDocument;
31     Document currentDoc = uiDoc.Document;
32
33     try {
34         // Find all walls instances in the document by using category filter!
35         ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);
36         // Apply the filter to the elements in the active document!
37         // Use shortcut WhereElementIsNotElementType() to find wall instances only
38         FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
39
40         IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
41
42         String prompt = "The walls in the current document are:\n";
43         foreach (Element e in walls) {
44             prompt = prompt + e.Name + "\n";
45         }
46         TaskDialog.Show("Revit", prompt);
47     } catch (Exception) {
48         throw;
49     }
50 }

```

Um laço “foreach” é utilizado para que cada item da coleção “walls” seja analisada, e nome do elemento é acumulado em uma variável de texto, uma string.

```

27 Revit Macros generated code
28
29 public void WallInfo(){
30     // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
31     UIDocument uiDoc = this.ActiveUIDocument;
32     Document currentDoc = uiDoc.Document;
33
34     try {
35         // Find all walls instances in the document by using category filter!
36         ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);
37         // Apply the filter to the elements in the active document!
38         // Use shortcut WhereElementIsNotElementType() to find wall instances only
39         FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
40
41         IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
42
43         String prompt = "The walls in the current document are:\n";
44         foreach (Element e in walls) {
45             prompt = prompt + e.Name + "\n";
46         }
47         TaskDialog.Show("Revit", prompt);
48     } catch (Exception) {
49         throw;
50     }
51 }
52
53
54
55
56
}

```

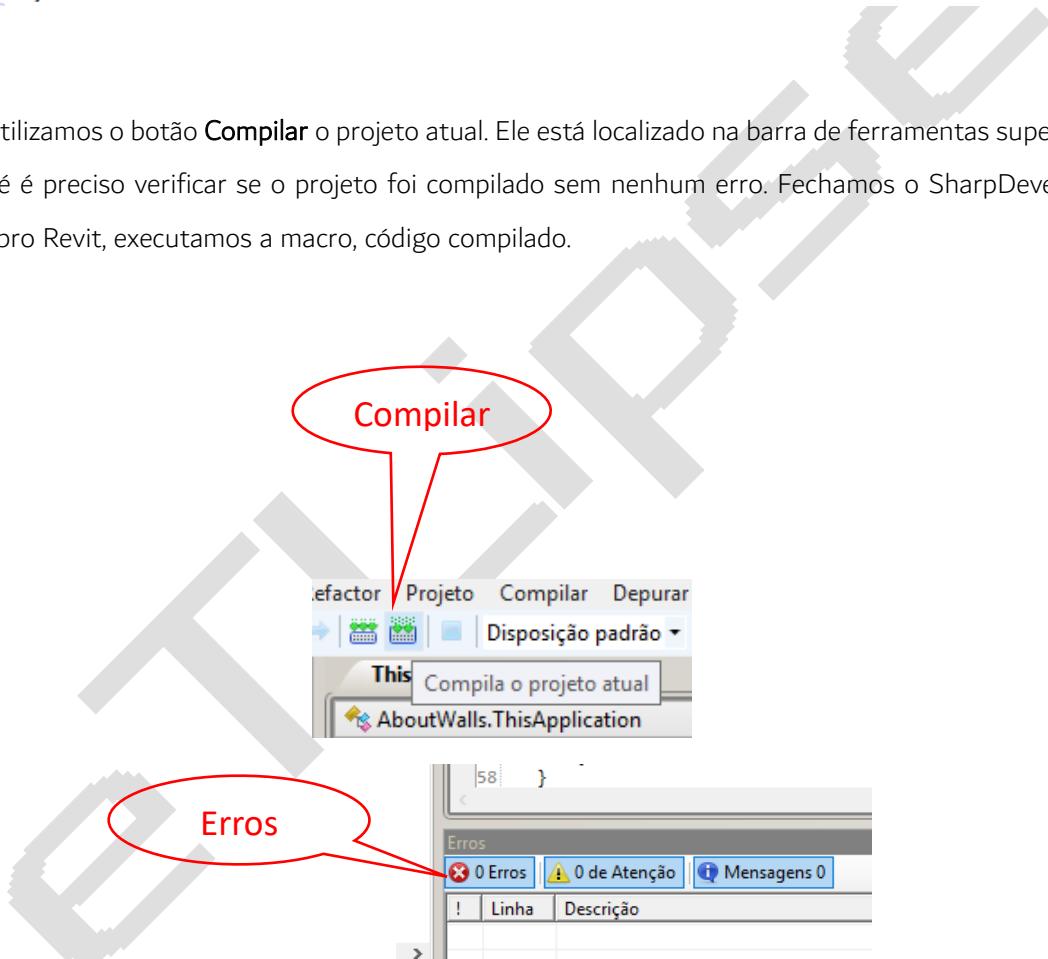
A string composta é finalmente exibida através de uma caixa de diálogo.

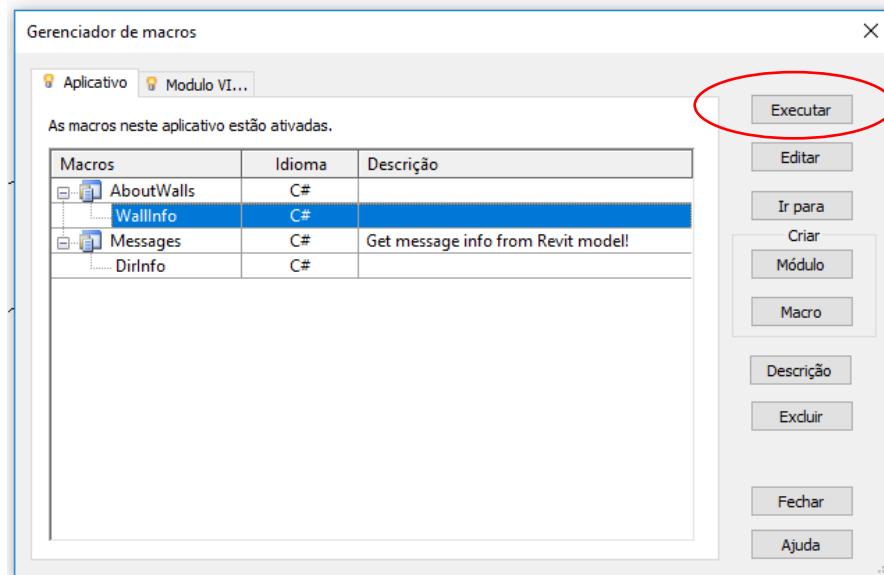
```

27 Revit Macros generated code
28
29 public void WallInfo(){
30     // Setup Revit UI (uiDoc) and Current Model (CurrentDoc)!
31     UIDocument uiDoc = this.ActiveUIDocument;
32     Document currentDoc = uiDoc.Document;
33
34     try {
35         // Find all walls instances in the document by using category filter!
36         ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);
37         // Apply the filter to the elements in the active document!
38         // Use shortcut WhereElementIsNotElementType() to find wall instances only
39         FilteredElementCollector collector = new FilteredElementCollector(currentDoc);
40
41         IList<Element> walls = collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
42
43         String prompt = "The walls in the current document are:\n";
44         foreach (Element e in walls) {
45             prompt = prompt + e.Name + "\n";
46         }
47         TaskDialog.Show("Revit", prompt);
48     } catch (Exception) {
49         throw;
50     }
51 }
52
53 }
54
55 }
56

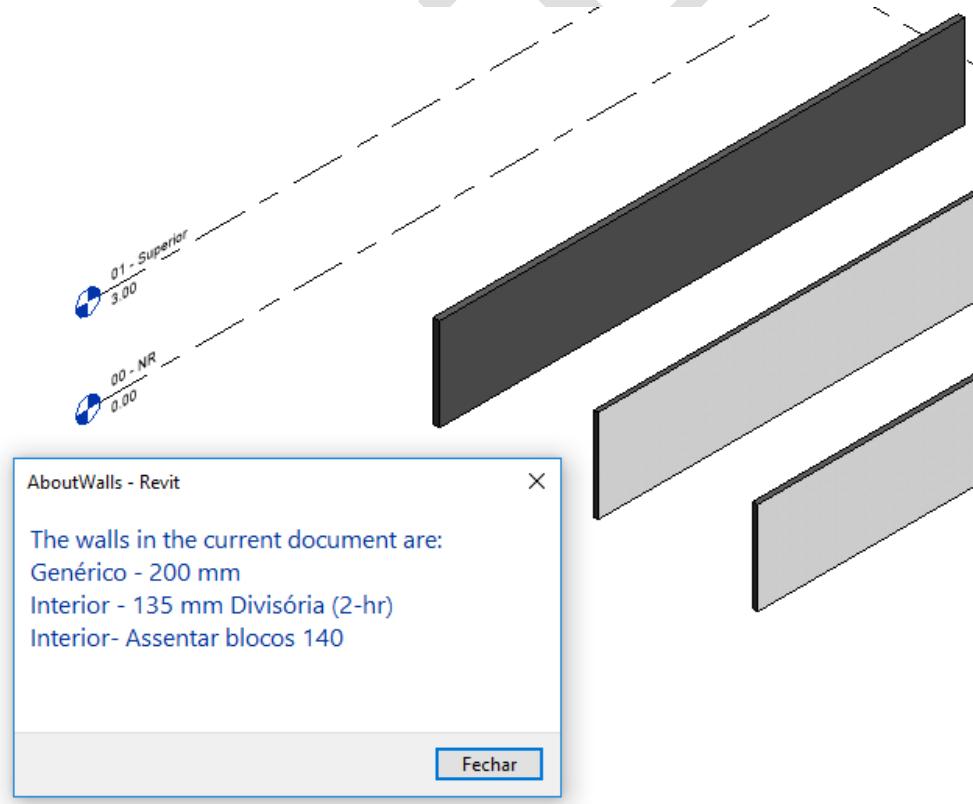
```

Utilizamos o botão **Compilar** o projeto atual. Ele está localizado na barra de ferramentas superior.
 No rodapé é preciso verificar se o projeto foi compilado sem nenhum erro. Fechamos o SharpDevelop.
 Voltanto pro Revit, executamos a macro, código compilado.



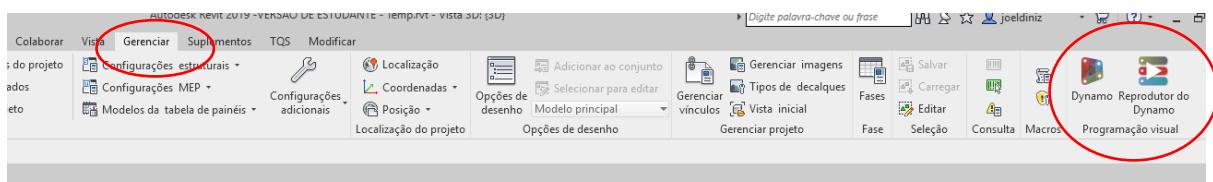


A aplicação é executada. As três paredes instanciadas neste modelo são acessadas. O nome das paredes é exibido em uma caixa de mensagem. Um botão Fechar permite que o programa seja finalizado.

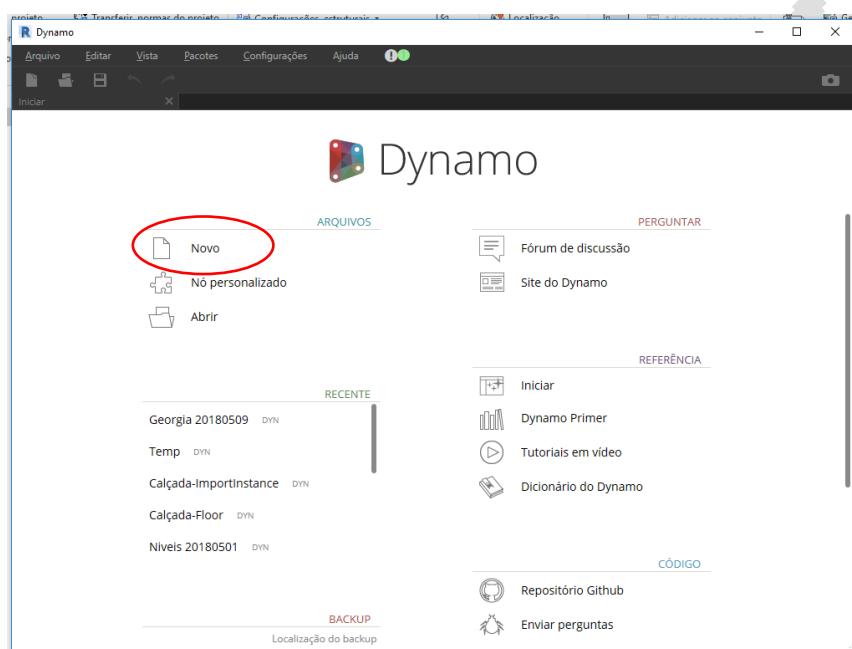


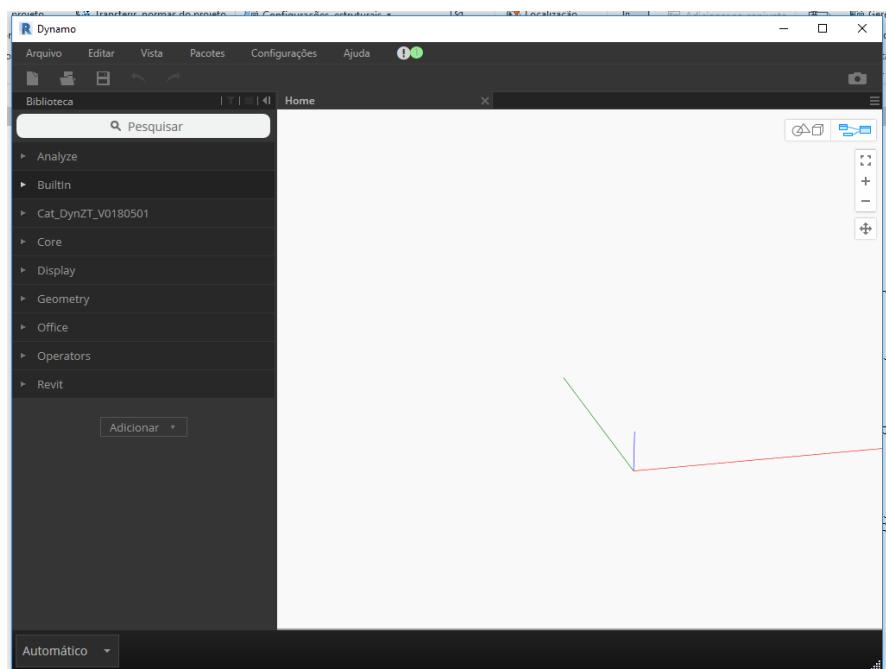
Programação Visual - Dynamo

O Dynamo pode ser acessado partindo da aba Gerenciar.



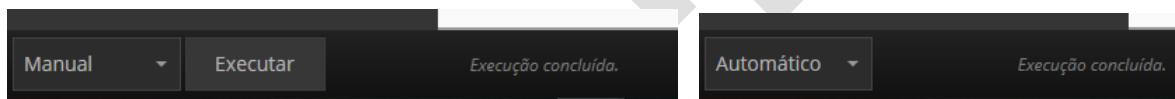
O Dynamo será inicializado e um novo projeto pode ser aberto.



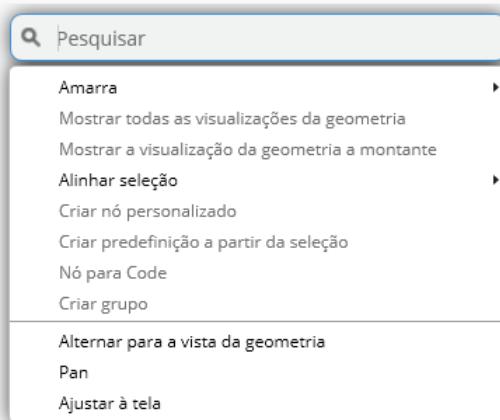


No Dynamo, no canto inferior esquerdo é possível selecionar duas formas de execução.

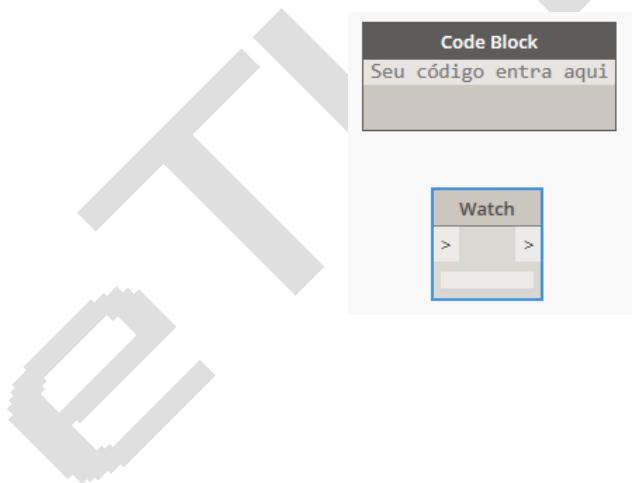
Automática, o Dynamo sempre irá rodar quando houver alteração no seu programa. Manual, será preciso clicar em Executar para o programa rodar. A recomendação é usar Manual em rotinas mais complexas, o que irá evitar maior consumo de recursos.



O Dynamo funciona com blocos lógicos, que possuem entrada e saída, sendo estas interligadas por conectores. Clicar com o botão direito do mouse na área de trabalho do Dynamo permite a pesquisa de blocos prontos a serem utilizados. Um bloco possui conectores de entrada e de saída. Os conectores devem ser interligados segundo lógica desejada.



Um bloco “Code Block” permite digitar código para ser executado no Dynamo. O código precisa ser escrito seguindo as regras da linguagem básica utilizada pelo Dynamo. Maiores detalhes sobre blocos para linguagem textual será tratado adiante em tópico específico para quem desejar usar este recurso com maior frequência, embora existam alternativas de nodes para diversas situações, como o uso de strings, que podem ser via linguagem textual, mas que possui node específico. A colocação de descrição breve deste recurso é para dar apenas noção para aqueles que não desejam usar tanto este recurso. Outro bloco muito útil é o “Watch”, usado para observar a saída de um módulo, ou uma sequência de módulos conectados.



É possível gerar sequências de números utilizando regras da linguagem base do Dynamo. Seguem exemplos:

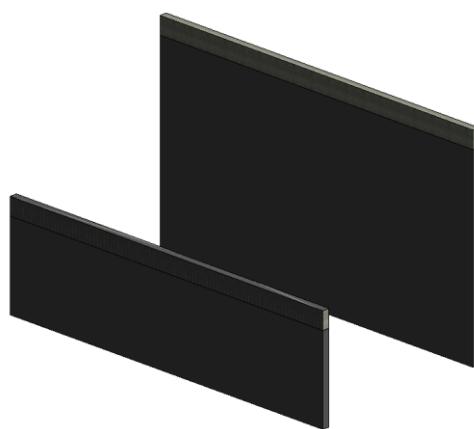
`a = 1..7;` Gera números de 1 a 7. O padrão é usar incremento inteiro. O incremento pode ser especificado (`a = 0..1..0.1`), mas pode não terminar no último número especificado (`a = 0..7..0.75`);

`a = 0..7..~0.75;` Garante a sequência com o último número especificado, usando incremento o mais próximo possível do especificado.

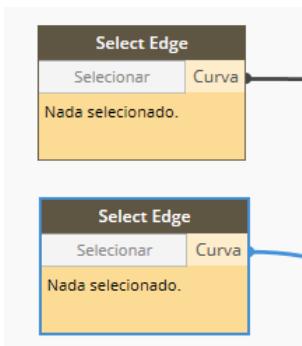
$a = 0..7..#9;$ Gera a sequência com o número especificado.

Exemplo Prático Transversinas

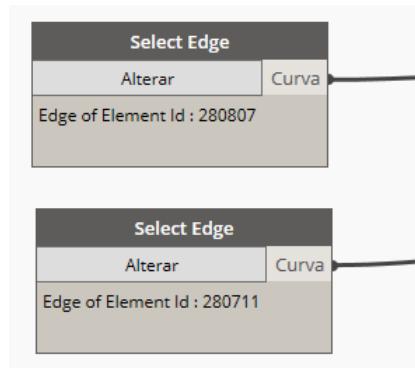
Objetivo deste exemplo é a colocação automática de transversinas sobre duas longarinas. Precisamos preparar uma estrutura para aplicar o exemplo. Criamos duas paredes com níveis diferentes. Colocamos uma longarina (viga) sobre cada uma delas.



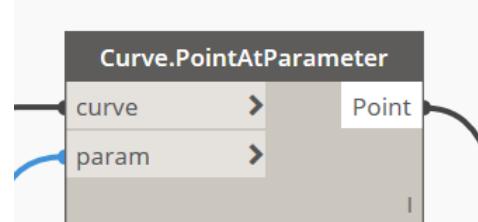
Iniciamos com o bloco “Select Edge”. Este módulo permite a seleção de uma instância que será utilizada como contorno. Precisamos de dois blocos, visto que usaremos as duas longarinas como condição de contorno. O bloco aparece amarelo e exibe o botão selecionar, enquanto os limites não foram selecionados.



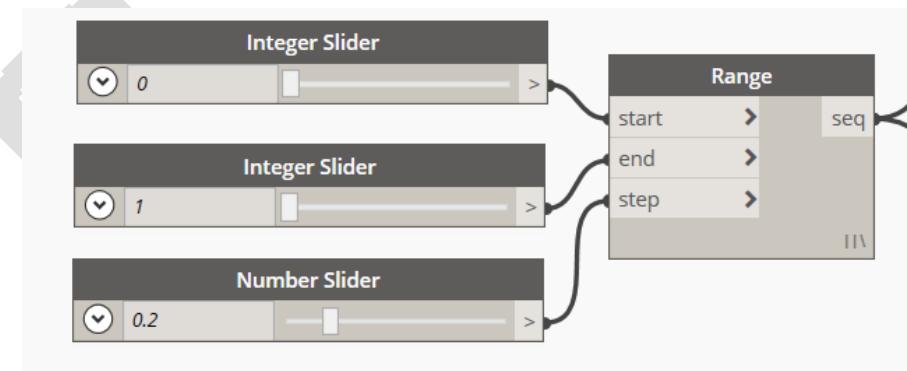
As duas longarinas devem ser selecionadas, uma por cada bloco. O bloco muda de cor. O ID de cada elemento é exibido no respectivo bloco no qual foi selecionado.

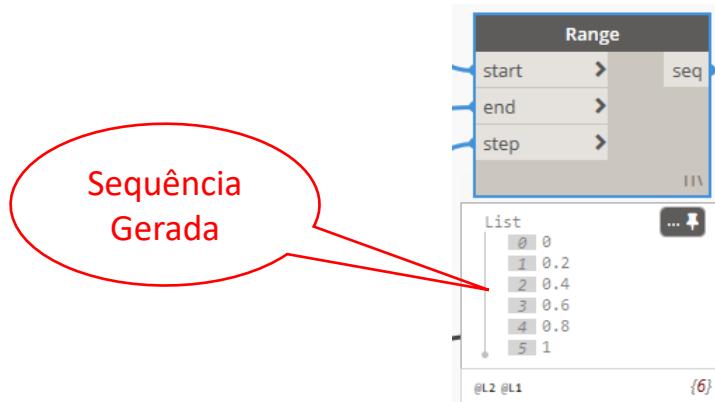


Utilizamos o bloco Curve.PointAtParameter para segmentar a linha base em partes iguais. Possuímos duas longarinas, logo teremos dois destes blocos. A divisão ocorre segundo o parâmetro de entrada em "param".

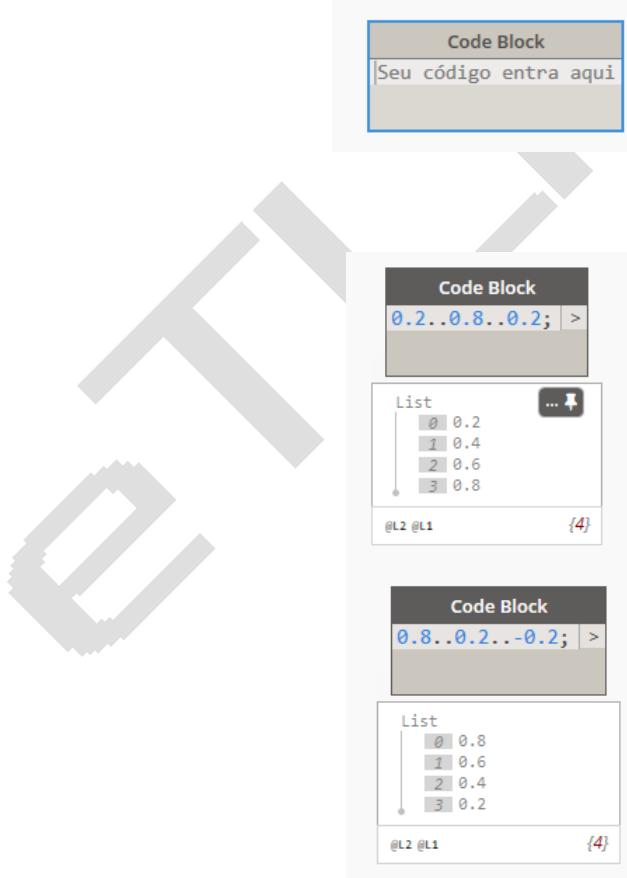


Utilizamos o bloco Range para fazer a composição da entrada do parâmetro. Utilizamos sliders para selecionar dois números inteiros e um real. Range recebe um número inicial, um incremento, e um número final. O inicial é incrementado até atingir o valor final.



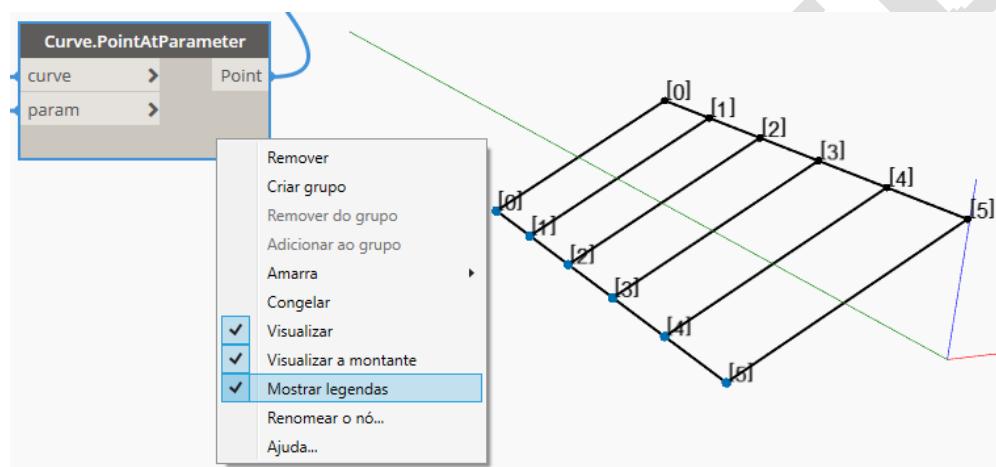


Pode ocorrer de os pontos não serem combinados corretamente, dependendo das linhas bases selecionadas. Uma alternativa é o uso de um bloco de código, Code Block, que pode ser ativado com duplo clique na área de trabalho do Dynamo. Use como entradas de dados “0.2..0.8..0.2;” e “0.8..0.2..-0.2;”. Outra opção é o uso do node List.Reverse.

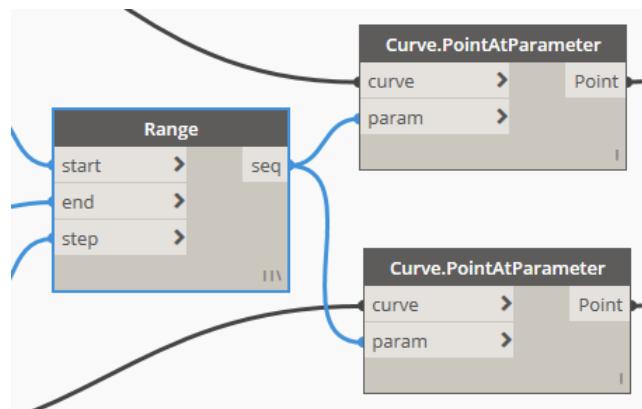




Clique com o botão direito do mouse no bloco Curve.PointAtParameter. Habilite a opção “Mostrar legendas”. É possível ver a legenda dos pontos definidos sobre as linhas bases, as longarinas selecionadas.

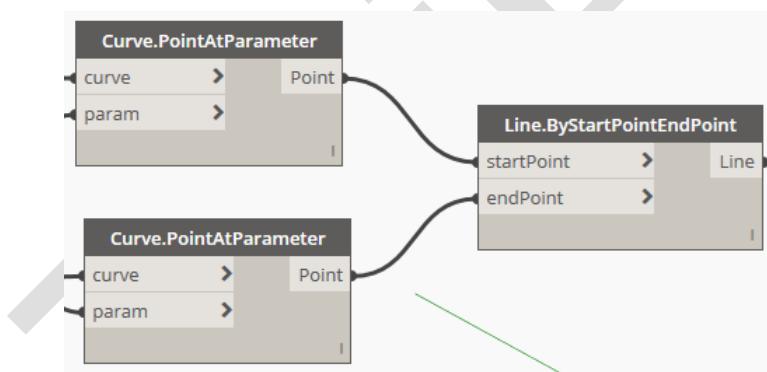


Curve.PointAtParameter será então imputado pelos parâmetros que definem a sequência de pontos e pelas curvas que irão orientar a direção dos pontos. Uma vez que as duas curvas serão seccionadas da mesma forma, é possível usar o mesmo bloco Range. A entrada das curvas virá de cada longarina selecionada.



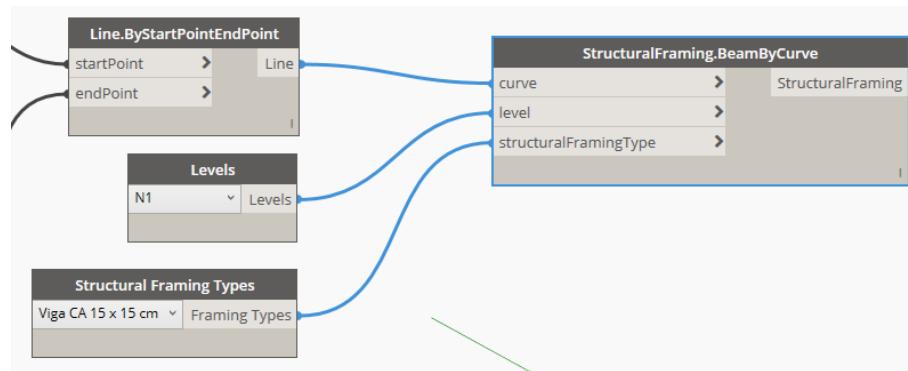
Utilize um bloco Line.ByStartEndPoint. Este bloco define uma linha com um ponto inicial e outro final.

Os pontos definidos sobre as longarinas devem ser utilizados para entrada de dados, que resultará nas linhas de base das transversinas.

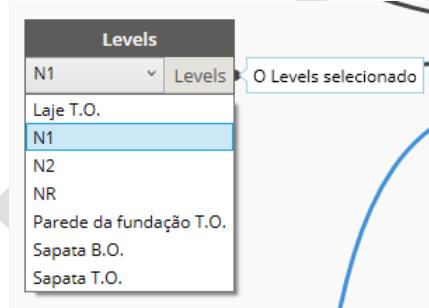


Utilize um bloco StructuralFraming.BeamByCurve. Este bloco possibilita a construção da transversina utilizando três parâmetros. O primeiro parâmetro é o anteriormente elaborado, que define as linhas bases das transversinas.

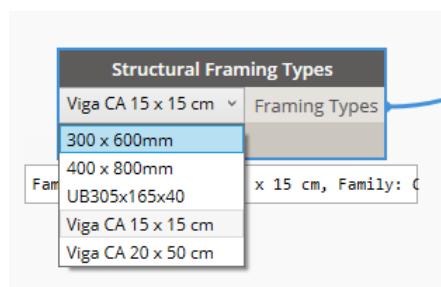




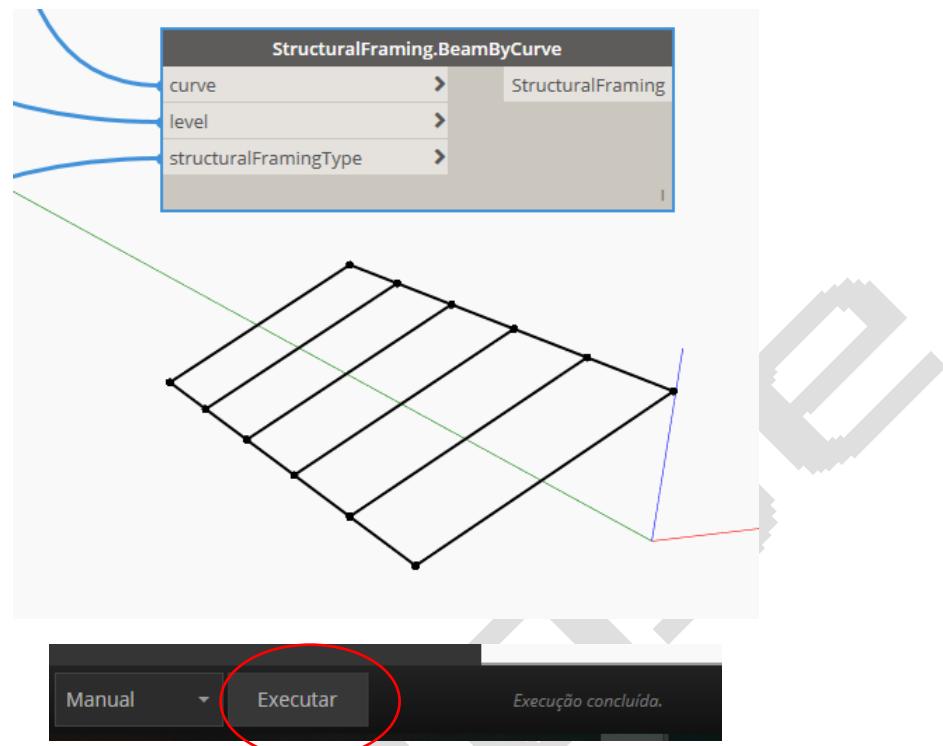
Utilize um bloco Levels. Este bloco possibilita selecionar qualquer um dos níveis já definidos no modelo ativo. O nível deve ser selecionado da lista disponibilizada por este bloco.



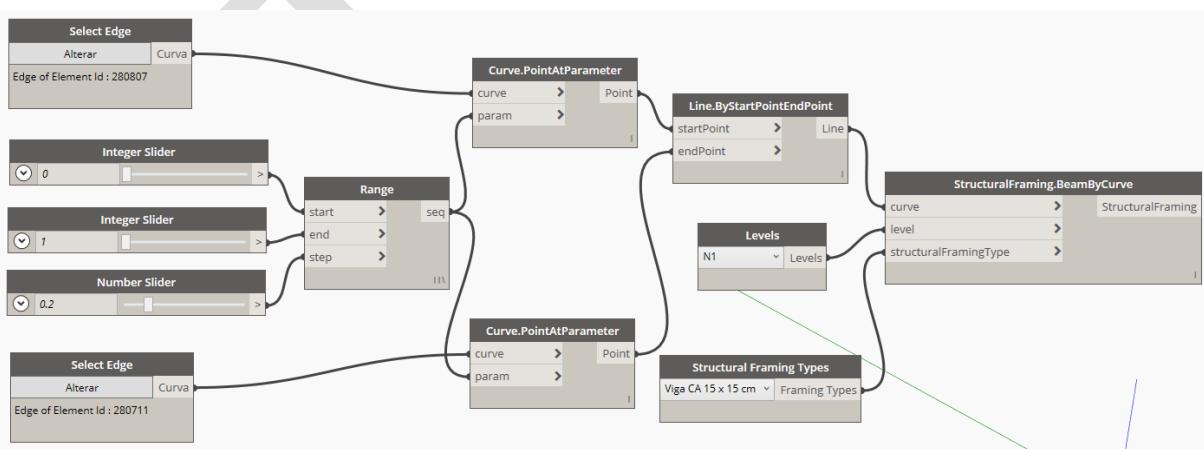
Utilize um bloco Structural Framing Types. Este bloco possibilita selecionar qualquer um dos suportes estruturais carregados no modelo ativo. A opção adequada deve ser selecionada, neste exemplo uma viga de Concreto Armado de 15 cm por 15 cm foi selecionada.



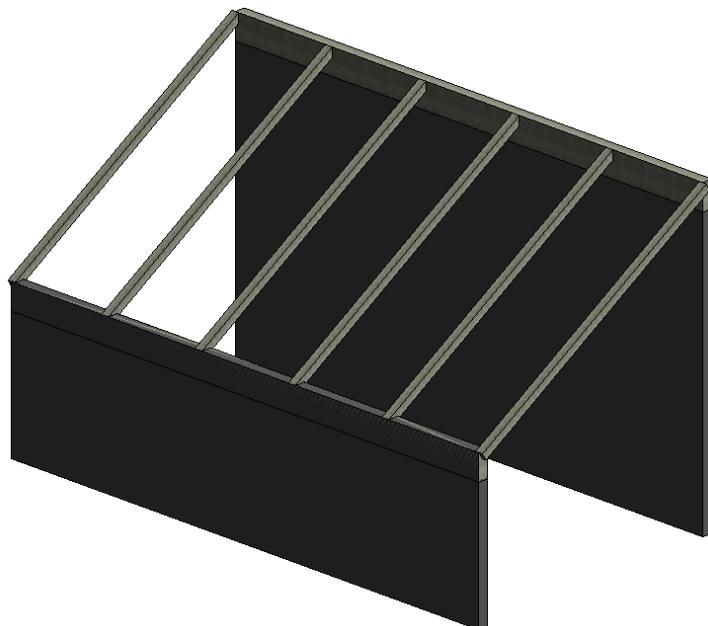
Finalmente o último bloco, StructuralFraming BeamByCurve, possui as entradas necessárias para a saída pretendida. Executamos a rotina. As longarinas são definidas e sua estrutura base já pode ser visualizada na própria área de trabalho do Dynamo.



Rotina completa no Dynamo para o exemplo prático das transversinas.

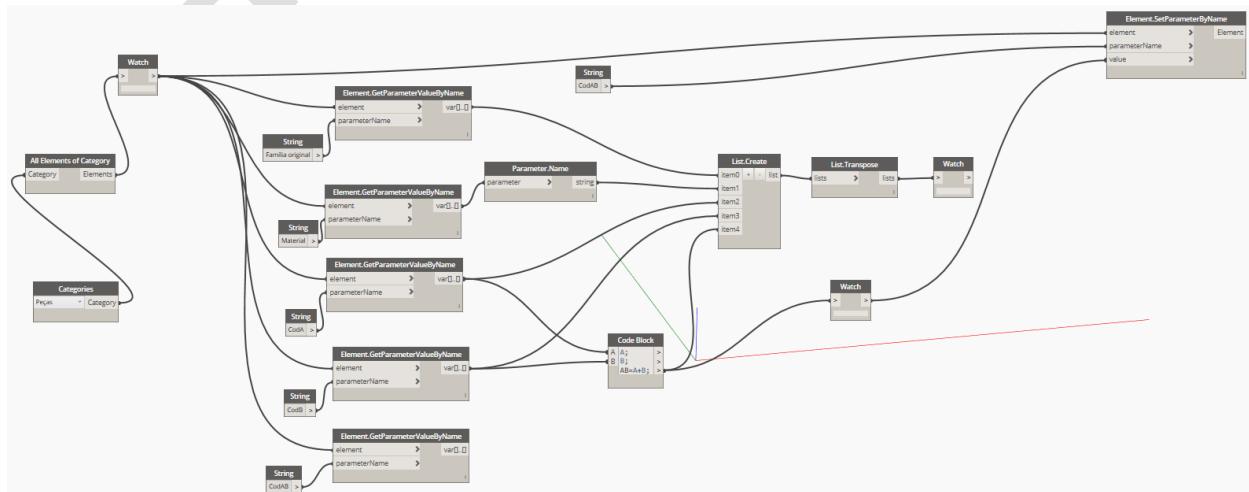


Resultado Final. Traversinas (vigotas) sobre as longarinas (vagas) selecionadas.



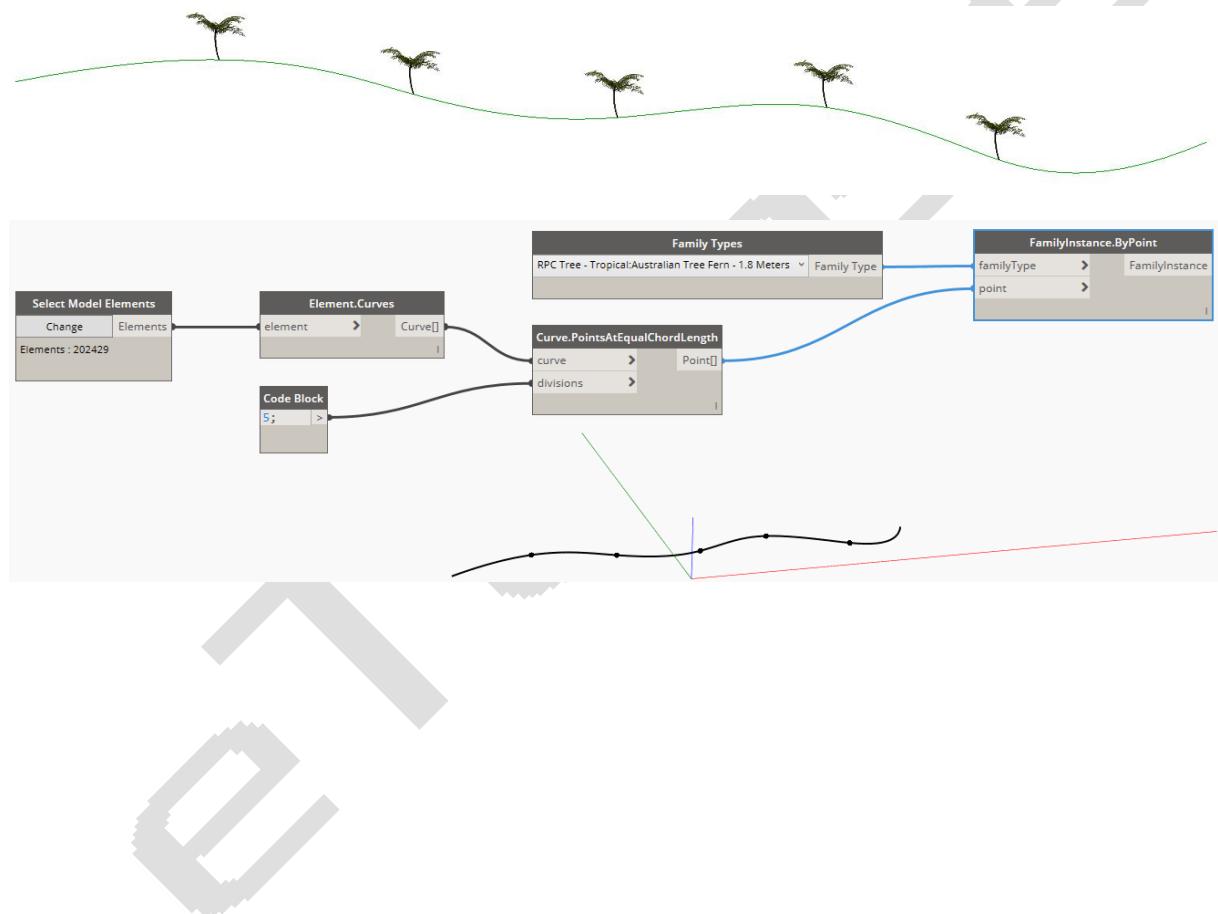
Exemplo - Codificação para BIM 4D

Rotina no Dynamo para exemplo prático de codificação, que facilita associar uma Estrutura Analítica de Projeto a um modelo BIM 3D, por exemplo usando o software NavisWorks, fluxo BIM 4D não contemplado neste material.



Exemplo - Cerca Viva

Este exemplo permite que coloquemos uma vegetação espaçada automaticamente ao longo de uma linha guia. Obviamente podemos selecionar outro tipo de elemento para estrutura da cerca, tal como um poste. A linha guia precisar ser criada no Revit para posteriormente ser selecionada, e o elemento a ser usado para repetição deve estar carregado no modelo.



Uso de Código no Dynamo

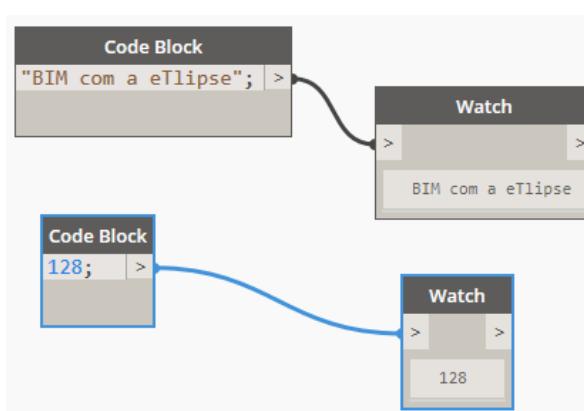
Embora o propósito maior do Dynamo obviamente seja o uso da linguagem visual, não raro precisamos fazer uso da linguagem textual. O Dynamo possui node dentro do qual podemos usar a linguagem textual, é o chamado Code Block. Nestes blocos usamos a linguagem base do Dynamo, que não será desafio para quem conhece C++ e suas derivadas, como C#. Com duplo clique na área de trabalho do Dynamo podemos incluir um node Code Block. Usaremos o Code Block acoplado com o node Watch, que nos possibilita ver a saída do que estamos digitando.

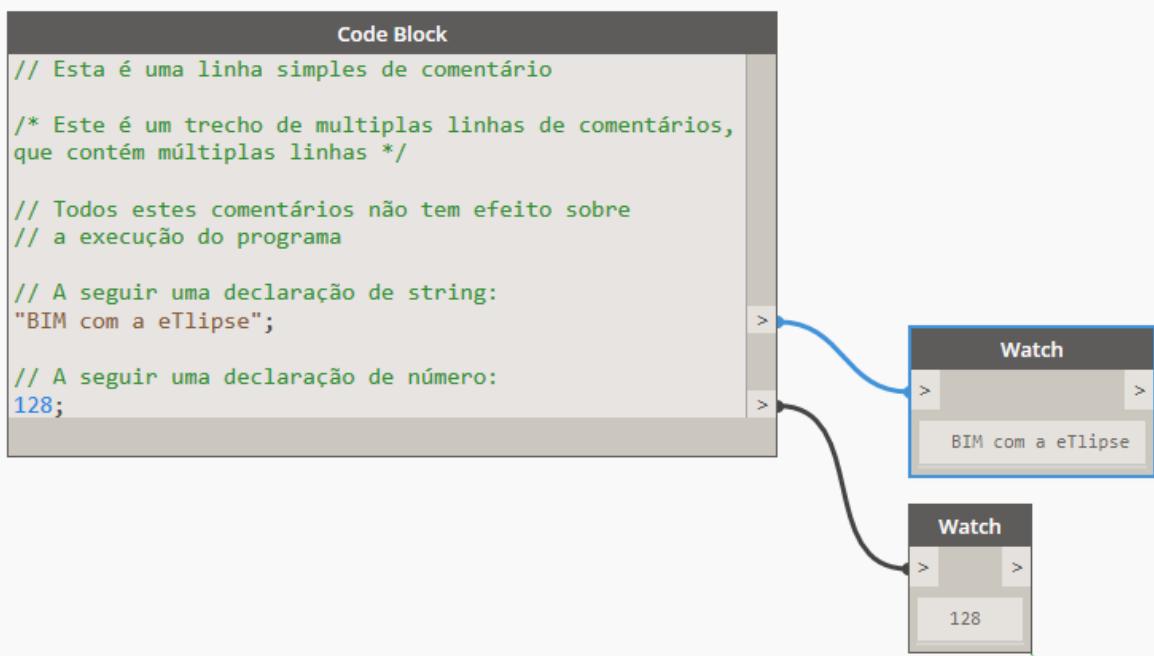
A sequência deste subcapítulo apresenta uma tradução do “Manual da Linguagem Dynamo”, que pode ser adquirido originalmente em Inglês. Os exemplos foram implementados para possibilitar uma visão composta do código e seu resultado, e os comentários foram traduzidos para o Português.



Linguagem Base

As coincidências com a linguagem C# não são mera coincidência! Os comandos devem terminar com ";" e podemos usar comentário de uma linha com "//" ou múltipla linhas com "/* */". Números devem ser digitados diretamente, mas strings devem estar entre aspas.

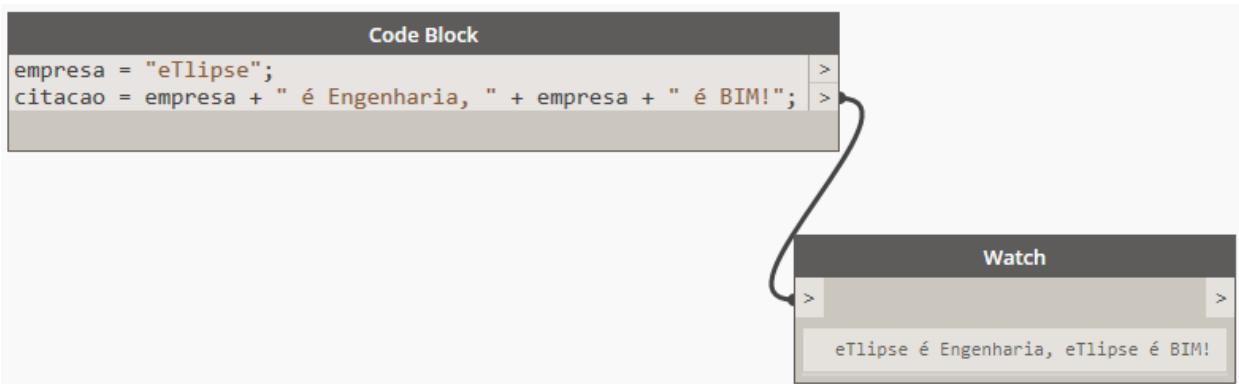




A declaração de variáveis se dá semelhante ao que utilizamos no C#. Importante não apenas atentar para as regras da linguagem, mas igualmente usar a boa prática de programação, como o uso de nomes lógicos e legíveis.

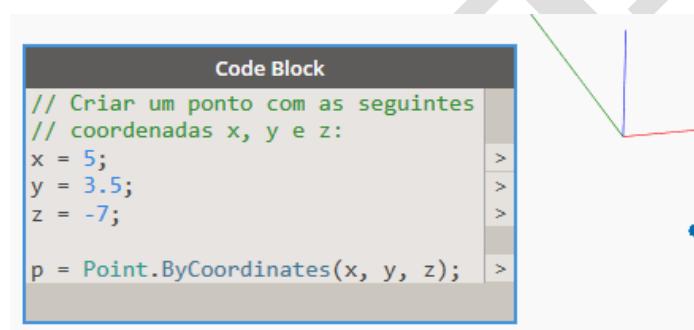


É possível concatenar strings com o uso de “+”, uma opção útil para gerar saídas que serão compostas em função de entradas variáveis.



Geometria Básica

O objeto geométrico mais básico na biblioteca padrão do Dynamo é o ponto. Toda geometria é instanciada a partir de construtores, obedecendo os tipos de classe da biblioteca. O construtor é utilizado com seu nome, no caso de ponto por “Point”, seguido por ponto e o método utilizado para instanciar o objeto. Para usar um ponto tridimensional usamos o método “ByCoordinates”.



Os construtores no Dynamo em geral usam o prefixo “By”, que retornam uma instância do objeto da classe utilizada. A instância é armazenada em uma variável do nome utilizado para receber a instância com o uso do sinal de atribuição “=”. Esta variável será do tipo da classe do construtor utilizado.

A maioria das classes possui vários tipos de construtores, por exemplo, ponto pode ser instanciado com coordenadas polares, usando raio da esfera, e dois ângulos de rotação, através do método “BySphericalCoordinates”.

Code Block

```
// Criar um ponto na esfera com os seguintes
// raio, angulos de rotação theta e phi (em graus):
radius = 5;
theta = 75.5;
phi = 120.3;
cs = CoordinateSystem.Identity();

p = Point.BySphericalCoordinates(cs, radius, theta, phi);
```

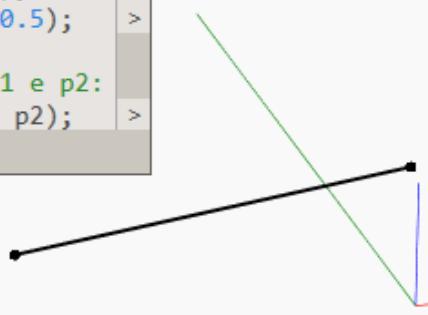


Pontos podem ser usados como base de construção para outras geometrias, como linhas, por exemplo. Podemos usar o construtor *ByStartPointEndPoint* para criar uma linha, *Line*, entre dois pontos.

Code Block

```
// Criar dois pontos:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

// Construir linha entre pontos p1 e p2:
l = Line.ByStartPointEndPoint(p1, p2);
```



Da mesma forma, linhas podem servir de base para construir superfícies geométricas complexas, por exemplo utilizando o construtor *Loft*, que pega uma série de linhas ou curvas e interpola uma superfície entre elas.

```

Code Block

// Criar dois pontos:
p1 = Point.ByCoordinates(3, 10, 2);
p2 = Point.ByCoordinates(-15, 7, 0.5);

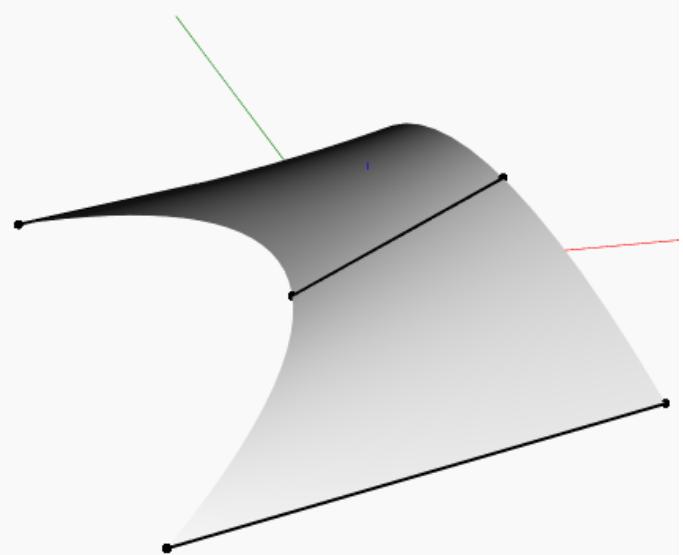
p3 = Point.ByCoordinates(5, -3, 5);
p4 = Point.ByCoordinates(-5, -6, 2);

p5 = Point.ByCoordinates(9, -10, -2);
p6 = Point.ByCoordinates(-11, -12, -4);

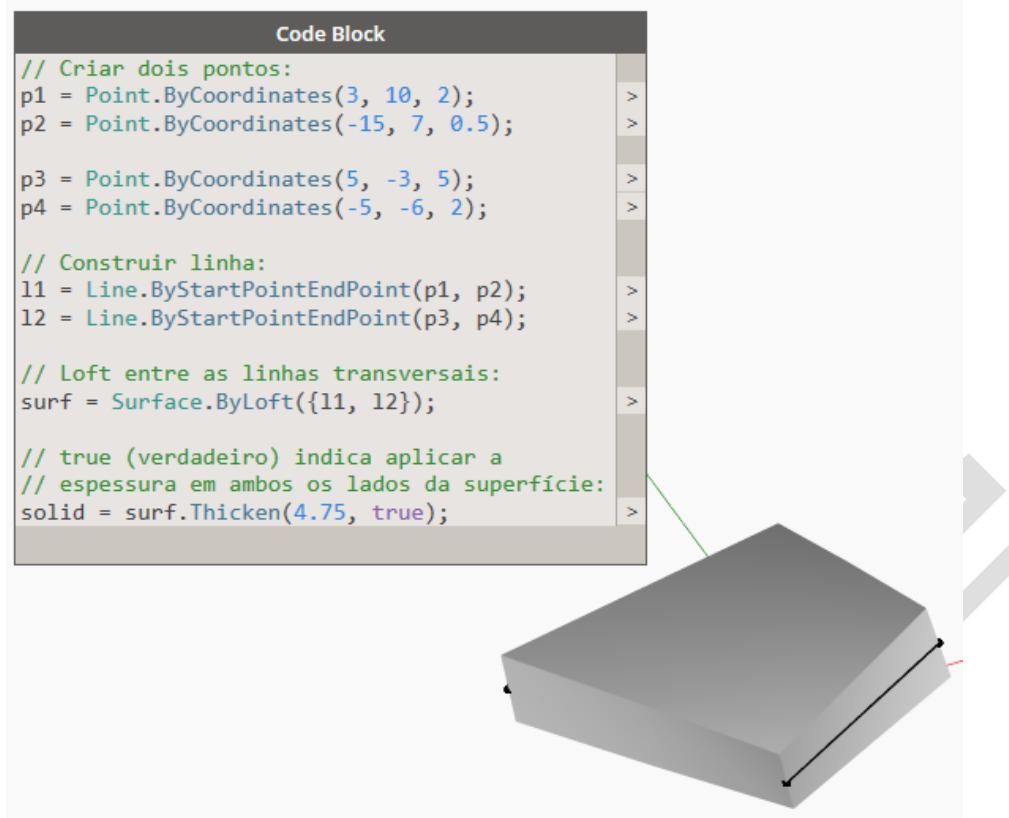
// Construir linha:
l1 = Line.ByStartPointEndPoint(p1, p2);
l2 = Line.ByStartPointEndPoint(p3, p4);
l3 = Line.ByStartPointEndPoint(p5, p6);

// Loft entre as linhas transversais:
surf = Surface.ByLoft({l1, l2, l3});

```

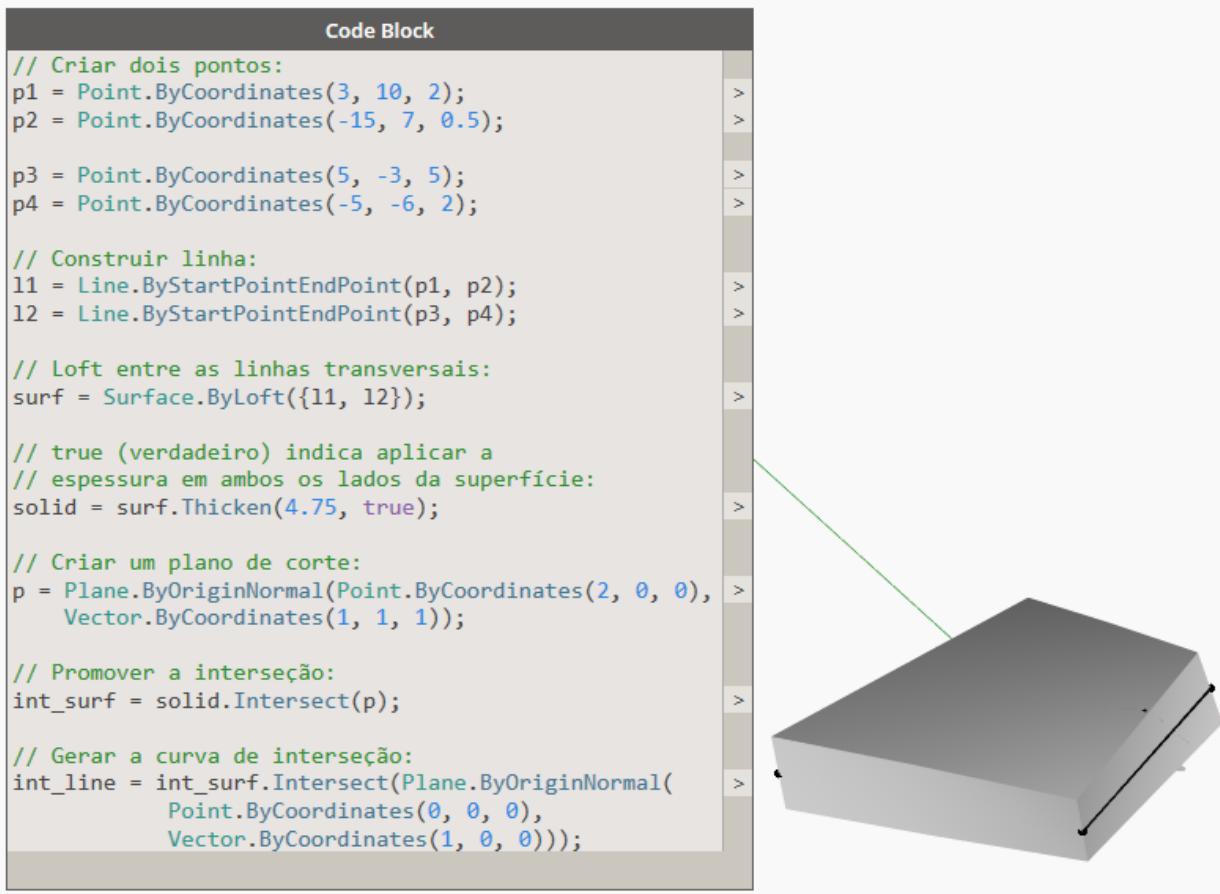


Superfícies também podem ser utilizadas para criar sólidos complexos, adicionando uma espessura à superfície poderemos obter um sólido. Muitos objetos possuem funções atreladas a eles, ao que se denomina métodos, permitindo ao programador executar comandos em dado objeto. Métodos comuns para todas as partes incluem translação e rotação, *Translate* e *Rotate*, que respectivamente translada (move) e rotacional a geometria por uma quantidade especificada. Superfícies possuem um método para espessura, *Thicken*, que usa uma entrada simples, um número que especifica a nova espessura da superfície.



Comandos de interseção podem extrair geometrias menos complexas de objetos mais complexos.

Essa geometria simples extraída pode ser a base para uma geometria complexa, em um processo cíclico de criação de geometria, extração e recriação. Neste exemplo, usamos um sólido gerado, *Solid*, para criar uma superfície, *Surface*, e usar a superfície para criar uma curva, *Curve*.

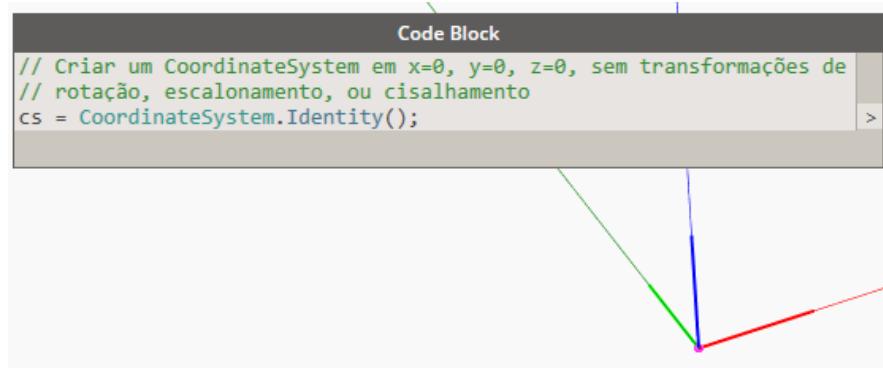


Geometrias Primitivas

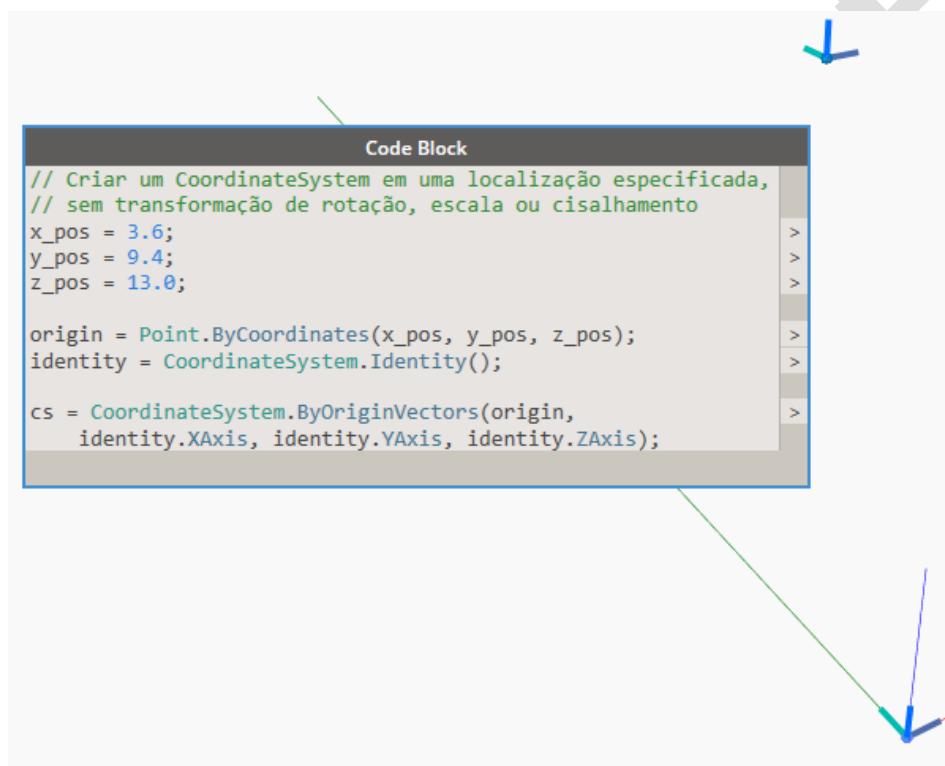
Enquanto Dynamo é capaz de criar uma variedade de formas geométricas complexas, geometrias primitivas simples formam a espinha dorsal de qualquer design computacional; ambos diretamente expressados no modelo final projetado, ou usados como armações das quais geometria mais complexa é gerada.

Embora não estritamente uma parte de geometria, o *CoordinateSystem* é uma ferramenta importante para construção geométrica. Um objeto *CoordinateSystem* mantém o rastro de ambos posição e transformações geométricas tais como rotação, cisalhamento e escala.

Criar um *CoordinateSystem* centralizado no ponto com $x=0$, $y=0$, $z=0$, com nenhuma transformação de rotação, escala, ou cisalhamento, simplesmente requer chamar o construtor identidade.



CoordinateSystem com transformação geométrica está além do escopo deste capítulo, contudo através de outro construtor é permitido criar um sistema de coordenadas em um ponto específico, *CoordinateSystem.ByOriginVectors*.



A geometria primitiva mais simples é um ponto, *Point*, representando um local zero-dimensional em um espaço tridimensional. Conforme já mencionado, existem diversas formas de criar um ponto em um sistema particular de coordenadas. *Point.ByCoordinates* cria um ponto com as coordenadas especificadas x, y e z. *Point.ByCartesianCoordinates* cria um ponto com coordenadas especificadas x, y e z em um sistema de coordenadas específico. *Point.ByCylindricalCoordinates* cria um ponto sobre um cilindro com raio, ângulo de rotação e altura. *Point.BySphericalCoordinates* cria um ponto sobre uma esfera com raio e dois ângulos de rotação. O exemplo a seguir mostra pontos criados por vários sistemas de coordenadas.

Code Block

```
// Criar um ponto com coordenadas x, y e z
x_pos = 1;
y_pos = 2;
z_pos = 3;

pCoord = Point.ByCoordinates(x_pos, y_pos, z_pos);

// Criar um ponto em um sistema de coordenadas específico
cs = CoordinateSystem.Identity();
pCoordinates = Point.ByCartesianCoordinates(cs, x_pos,
    y_pos, (z_pos + 1));
    // seguinte
// Criar um ponto em um cilindro com o seguinte
// raio e altura
radius = 5;
height = 15;
theta = 75.5;

pCyl = Point.ByCylindricalCoordinates(cs, radius,
    theta, height);

// Criar um ponto em uma esfera com raio e dois ângulos
phi = 120.3;

pSphere = Point.BySphericalCoordinates(cs, radius,
    theta, phi);
```

Point

A próxima primitiva do Dynamo é um segmento de linha, representando um número infinito de pontos entre dois pontos. Linhas podem ser criadas explicitamente estabelecendo os dois pontos limites com o construtor *Line.ByStartPointEndPoint*, ou especificando um ponto inicial, direção e comprimento nesta direção, *Line.ByStartPointDirectionLength*.

Code Block

```
// Criar dois pontos
p1 = Point.ByCoordinates(-2, -5, -10);
p2 = Point.ByCoordinates(6, 8, 10);

// Um segmento de linha entre dois pontos
l2pts = Line.ByStartPointEndPoint(p1, p2);

// Um segmento de linha em p1, na direção
// 1, 1, 1 com comprimento 10
lDir = Line.ByStartPointDirectionLength(p1,
    Vector.ByCoordinates(1, 1, 1), 10);
```

Dynamo tem objetos representando os tipos de primitivas geométricas mais básicos em três dimensões. Cuboides criados com *Cuboid.ByLengths*. Cones, criados com *Cone.ByPointsRadius* e *Cone.ByPointsRadii*. Cilindros, criados com *Cylinder.ByRadiusHeight*. Esferas, criadas com *Sphere.ByCenterPointRadius*.

Code Block

```
// Criar um cuboide com comprimento especificado
cs = CoordinateSystem.Identity();

cub = Cuboid.ByLengths(cs, 5, 15, 2);

// Criar diversos cones
p1 = Point.ByCoordinates(0, 0, 10);
p2 = Point.ByCoordinates(0, 0, 20);
p3 = Point.ByCoordinates(0, 0, 30);

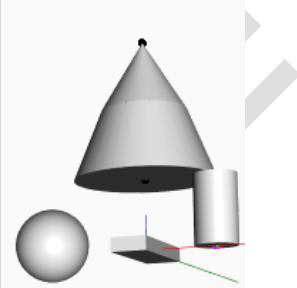
cone1 = Cone.ByPointsRadii(p1, p2, 10, 6);
cone2 = Cone.ByPointsRadii(p2, p3, 6, 0);

// Fazer um cilindro
cylcs = cs.Translate(10, 0, 0);

cyl = Cylinder.ByRadiusHeight(cylcs, 3, 10);

// Fazer uma esfera
centerP = Point.ByCoordinates(-10, -10, 0);

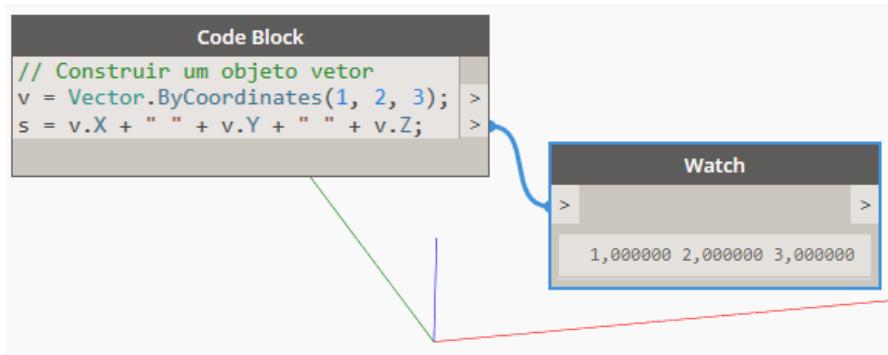
sph = Sphere.ByCenterPointRadius(centerP, 5);
```



Matemática de Vetor

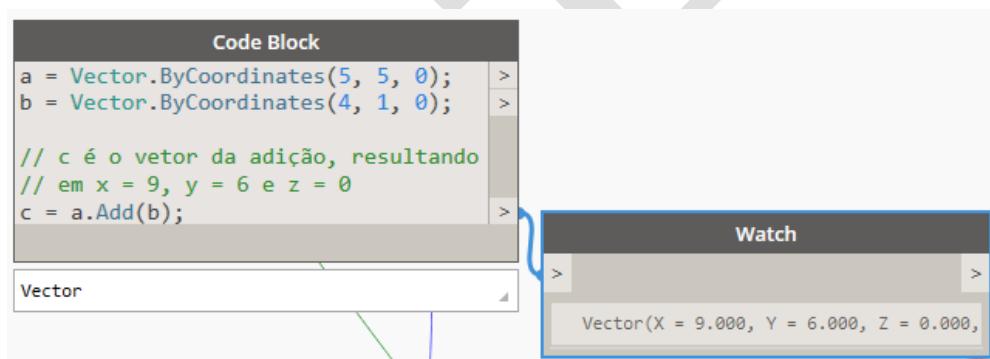
Objetos em projetos computacionais são raramente criados explicitamente em sua posição e forma final, e mais comumente são transladados, rotacionados, e de outra forma posicionados baseados fora de geometrias existentes. Matemática de vetores serve como um tipo de geometria andaim para dar direção e orientação para geometria, assim como para conceituar movimentos através do espaço 3D sem representação visual.

Como o mais básico, um vetor representa uma posição no espaço 3D, e é muitas vezes através do ponto final e uma seta a partir do ponto (0, 0, 0) para esta posição. Vetores podem ser criados com o construtor *ByCoordinates*, pegando a posição x, y e z do novo objeto *Vector* criado. Note que objetos *Vector* não são objetos geométricos, e não aparecem na janela do Dynamo. Contudo, informações sobre o novo vetor criado ou modificado pode ser impressas no janela console.



Um conjunto de operações matemáticas é definido nos objetos *Vector*, permitindo adicionar, subtrair, multiplicar e mover objetos no espaço 3D, como você moveria números reais no espaço 1D em uma linha numérica.

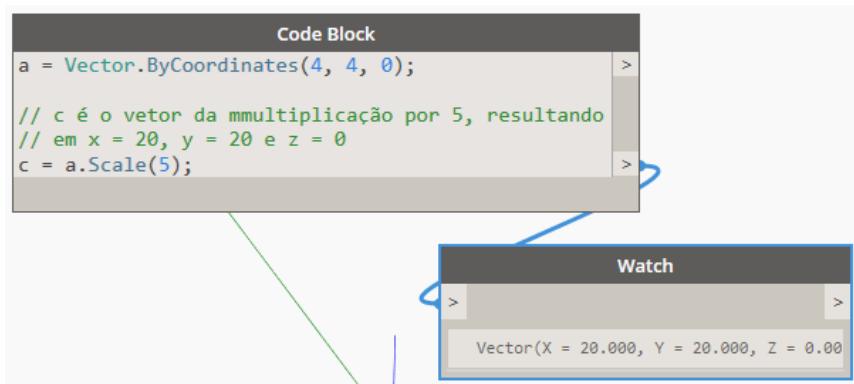
A adição de vetor é definida como a soma dos componentes de dois vetores e pode ser considerada o vetor resultante se as setas do vetor de dois componentes forem colocadas “de ponta a ponta”. A adição de vetor é realizada com o método *Add* e é representada por o diagrama à esquerda.



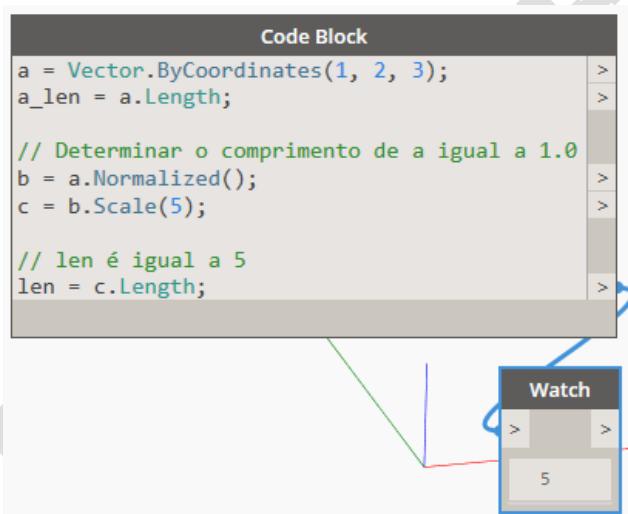
Da mesma forma, dois objetos *Vector* podem ser subtraídos um do outro com o método *Subtract*. A subtração do vetor pode ser considerada a direção do primeiro vetor para o segundo vetor.



A multiplicação de vetores pode ser considerada como mover o ponto final de um vetor em sua própria direção por um determinado fator de escala.

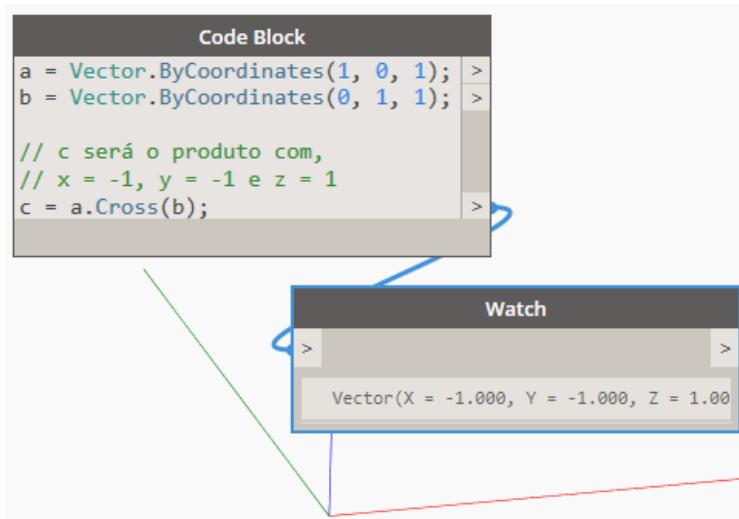


Muitas vezes, ao dimensionar um vetor, é desejável que o comprimento resultante seja exatamente igual à quantidade dimensionada. Isso é facilmente alcançado pela normalização de um vetor, em outras palavras, definindo o comprimento do vetor exatamente igual a um.

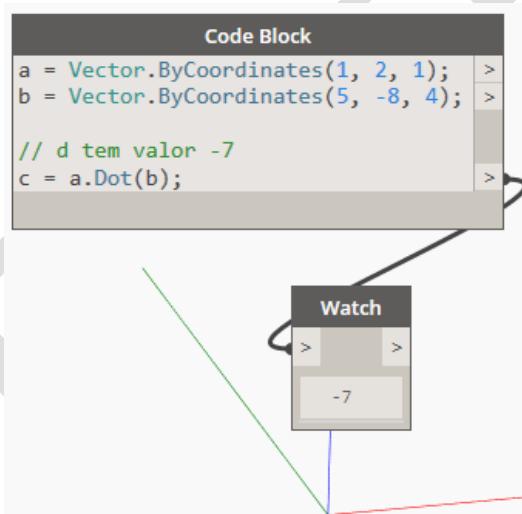


c ainda aponta na mesma direção que a $(1, 2, 3)$, embora agora tenha comprimento exatamente igual a 5.

Existem dois métodos adicionais na matemática vetorial, que não têm paralelos claros com a matemática 1D, o produto cruzado e o produto escalar. O produto cruzado é um meio de gerar um vetor ortogonal (de 90 graus a) a dois vetores existentes. Por exemplo, o produto cruzado dos eixos x e y é o eixo z, embora os dois vetores de entrada não precisem ser ortogonais entre si. Um vetor de produto cruzado é calculado com o método Cross.



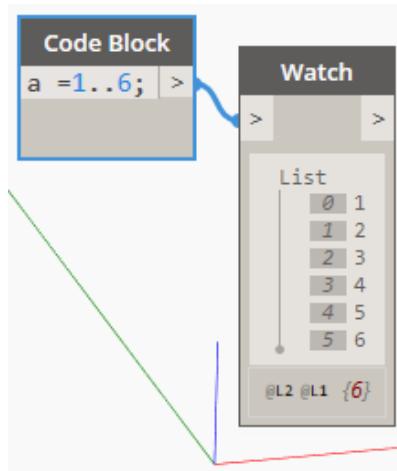
Uma função adicional, embora um pouco mais avançada da matemática vetorial, é o produto escalar. O produto escalar entre dois vetores é um número real (não um objeto *Vector*) que se relaciona, mas não é exatamente, o ângulo entre dois vetores. Uma propriedade útil do produto pontual é que o produto pontual entre dois vetores será 0 se e somente se forem perpendiculares. O produto escalar é calculado com o método *Dot*.



Expressões de Intervalo

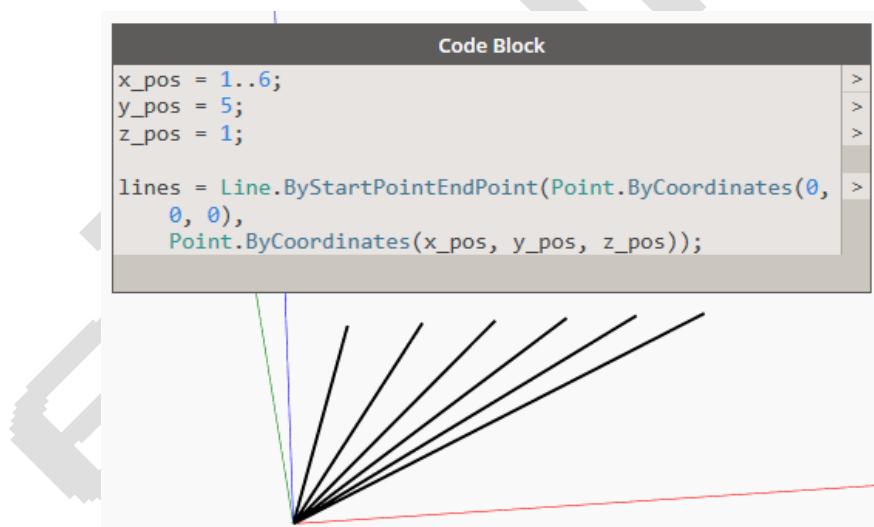
Quase todo design envolve elementos repetitivos, e digitar explicitamente os nomes e construtores de cada Ponto, Linha e outras primitivas em um script consumiria um tempo inviável. As expressões de intervalo fornecem ao programador Dynamo os meios para expressar conjuntos de valores como parâmetros em ambos os lados de dois pontos (..), gerando números intermediários entre esses dois extremos.

Por exemplo, enquanto vimos variáveis contendo um único número, é possível, com expressões de intervalo, ter variáveis que contêm um conjunto de números. A expressão de intervalo mais simples preenche os incrementos de número inteiro entre o início e o fim do intervalo.



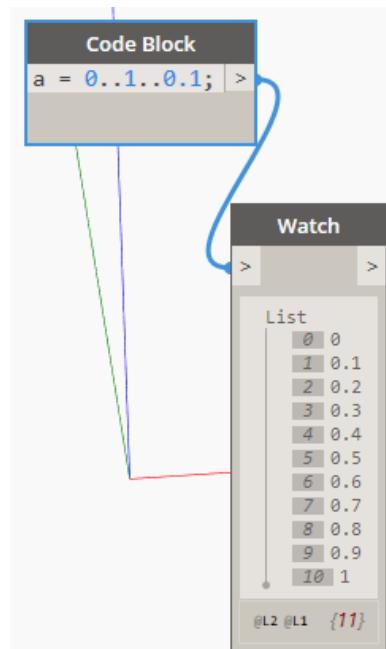
Nos exemplos anteriores, se um único número for passado como argumento de uma função, produziria um único resultado. Da mesma forma, se um intervalo de valores é passado como argumento de uma função, um intervalo de valores é retornado.

Por exemplo, se passarmos um intervalo de valores para o construtor *Line*, o Dynamo retornará um intervalo de linhas.

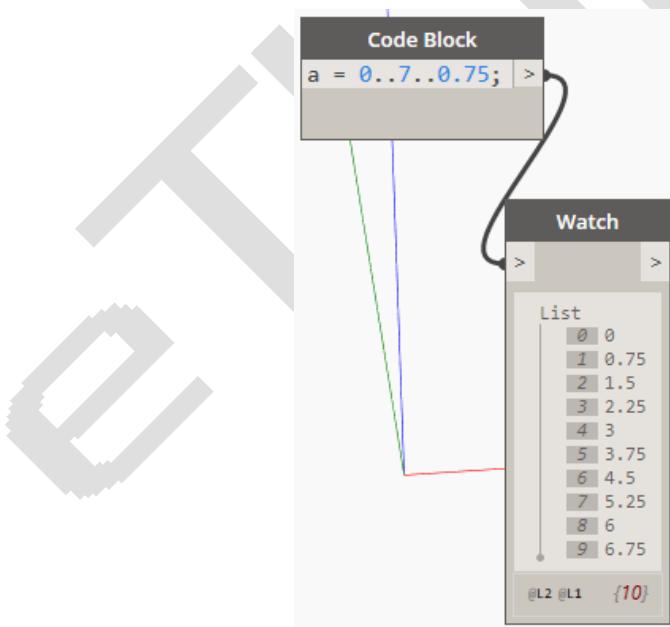


Por padrão, as expressões de intervalo preenchem o intervalo entre números incrementados por números inteiros, o que pode ser útil para um esboço topológico rápido, mas é menos apropriado para projetos reais. Adicionando um segundo dois pontos (...) à expressão de intervalo, você pode especificar a

quantidade que a expressão de intervalo incrementa entre valores. Aqui queremos todos os números entre 0 e 1, incrementando em 0,1:



Um problema que pode surgir ao especificar o incremento entre os limites da expressão de intervalo é que os números gerados nem sempre caem no valor final do intervalo. Por exemplo, se criarmos uma expressão de intervalo entre 0 e 7, incrementando em 0,75, os seguintes valores serão gerados:



Se um design exigir que uma expressão de intervalo gerada termine precisamente no valor máximo da expressão de intervalo, o Dynamo poderá aproximar um incremento, chegando o mais próximo possível, mantendo uma distribuição igual de números entre os limites do intervalo. Isso é feito com o sinal aproximado (~) antes do terceiro parâmetro:

Code Block

```
// DesignScript irá incrementar por 0.777 não 0.75
a = 0..7..~0.75;
```

Watch

	List
0	0
1	0.7777777777777778
2	1.55555555555556
3	2.33333333333333
4	3.11111111111111
5	3.88888888888889
6	4.66666666666667
7	5.44444444444444
8	6.22222222222222
9	7

{10}

No entanto, se você deseja que o Dynamo interpole entre intervalos com um número discreto de elementos, o operador # permite especificar isso:

Code Block

```
// Interpolar entre 0 e 7 de tal forma que
// "a" irá conter 9 elementos
a = 0..7..#9;
```

Watch

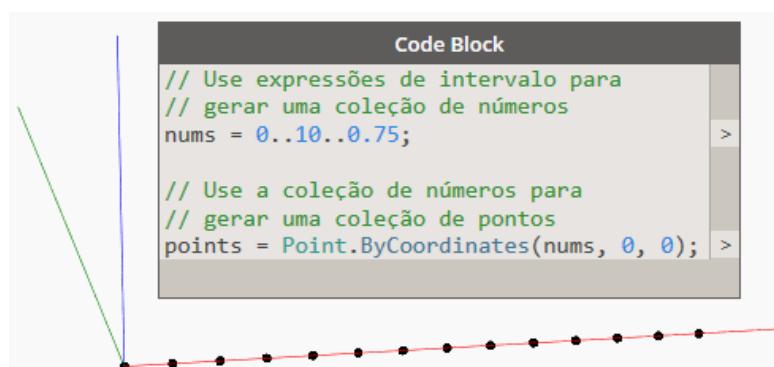
	List
0	0
1	0.875
2	1.75
3	2.625
4	3.5
5	4.375
6	5.25
7	6.125
8	7

{9}

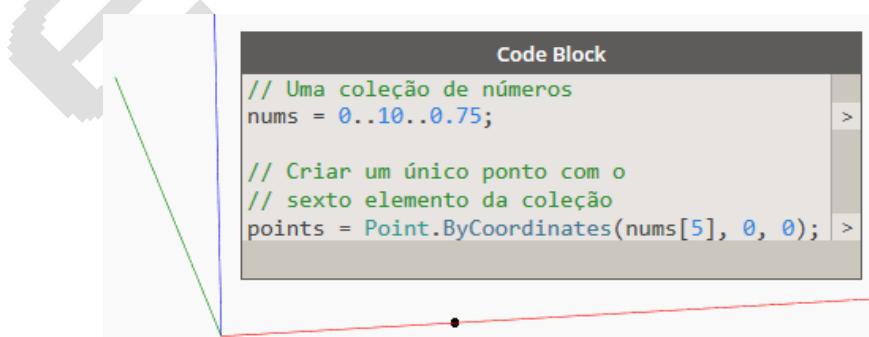
Coleções

Coleções são tipos especiais de variáveis que contêm conjuntos de valores. Por exemplo, uma coleção pode conter os valores 1 a 10, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], geometria variada a partir do resultado de uma operação de interseção, {Surface, Point, Line, Point} ou mesmo um conjunto de coleções, [[1, 2, 3], [4, 5], 6].

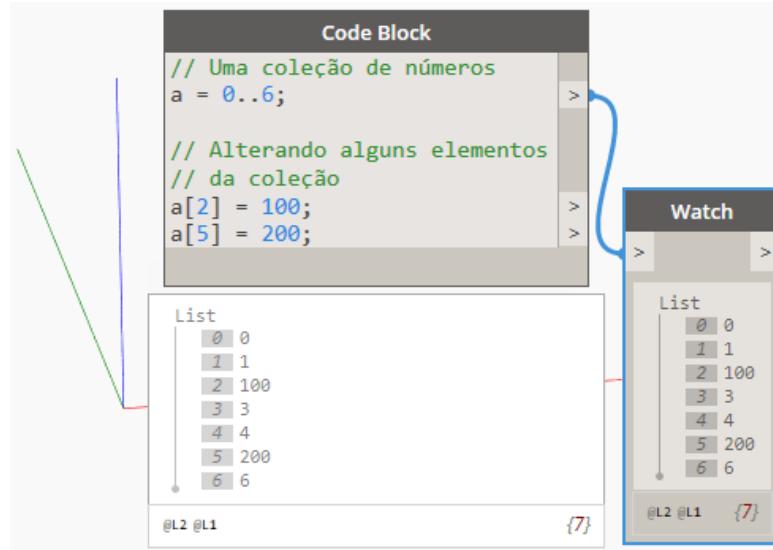
Uma das maneiras mais fáceis de gerar uma coleção é com expressões de intervalo (consulte: Expressões de intervalo). As expressões de intervalo, por padrão, geram coleções de números, embora se essas coleções forem passadas para funções ou construtores, as coleções de objetos serão retornadas.



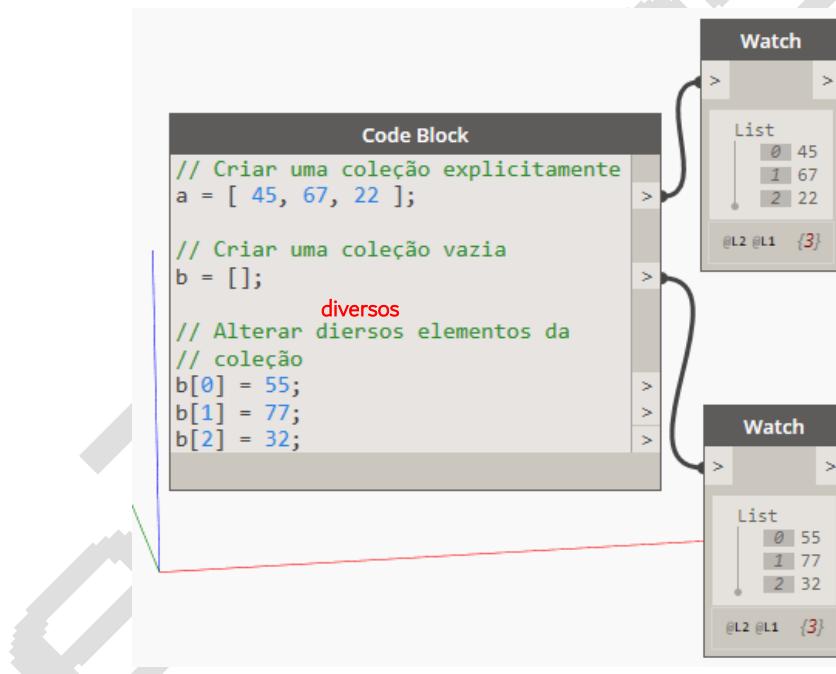
Quando as expressões de intervalo não são apropriadas, as coleções podem ser criadas vazias e preenchidas manualmente com valores. O operador de colchete ([]) é usado para acessar membros dentro de uma coleção. Os colchetes são escritos após o nome da variável, com o número do membro da coleção individual contido. Esse número é chamado de índice do membro da coleção. Por razões históricas, a indexação começa em 0, significando que o primeiro elemento de uma coleção é acessado com: coleção [0] e é frequentemente chamado de número "zeroth". Os membros subsequentes são acessados aumentando o índice em um, por exemplo:



Os membros individuais de uma coleção podem ser modificados usando o mesmo operador de índice após a criação da coleção:

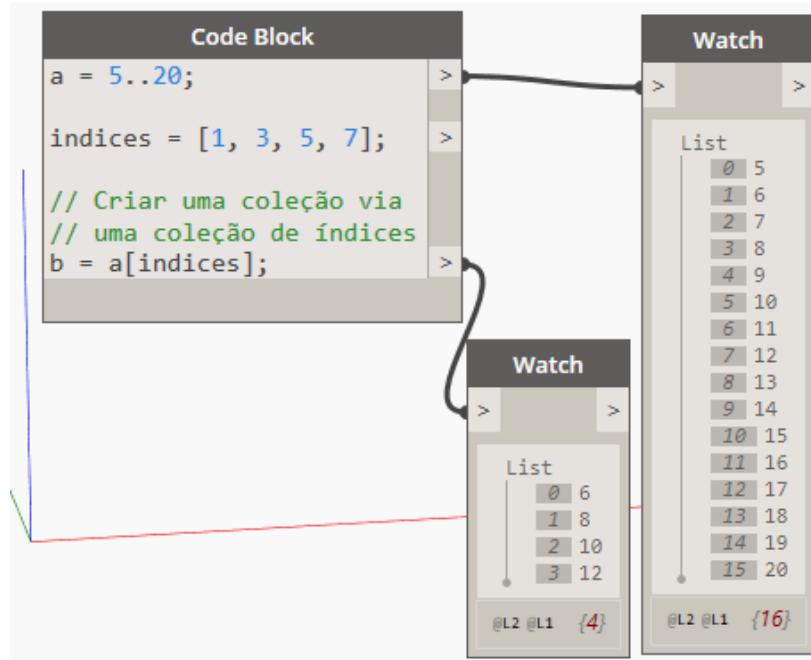


De fato, uma coleção inteira pode ser criada configurando explicitamente cada membro da coleção individualmente. Coleções explícitas são criadas com o operador de chaves ({}{}) envolvendo os valores iniciais da coleção ou deixadas em branco para criar uma coleção vazia:

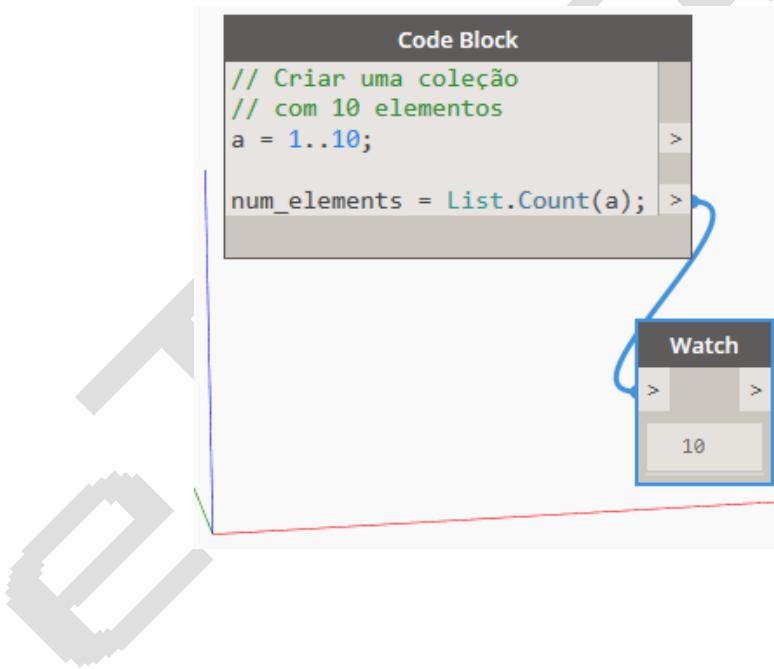


Coleções também podem ser usadas como índices para gerar novas sub-coleções de uma coleção.

Por exemplo, uma coleção contendo os números [1, 3, 5, 7], quando usada como índice de uma coleção, extrairia os 2º, 4º, 6º e 8º elementos de uma coleção (lembre-se de que os índices começam em 0):



O Dynamo contém funções utilitárias para ajudar a gerenciar coleções. A função *List.Count*, como o nome indica, conta uma coleção e retorna o número de elementos que ela contém.

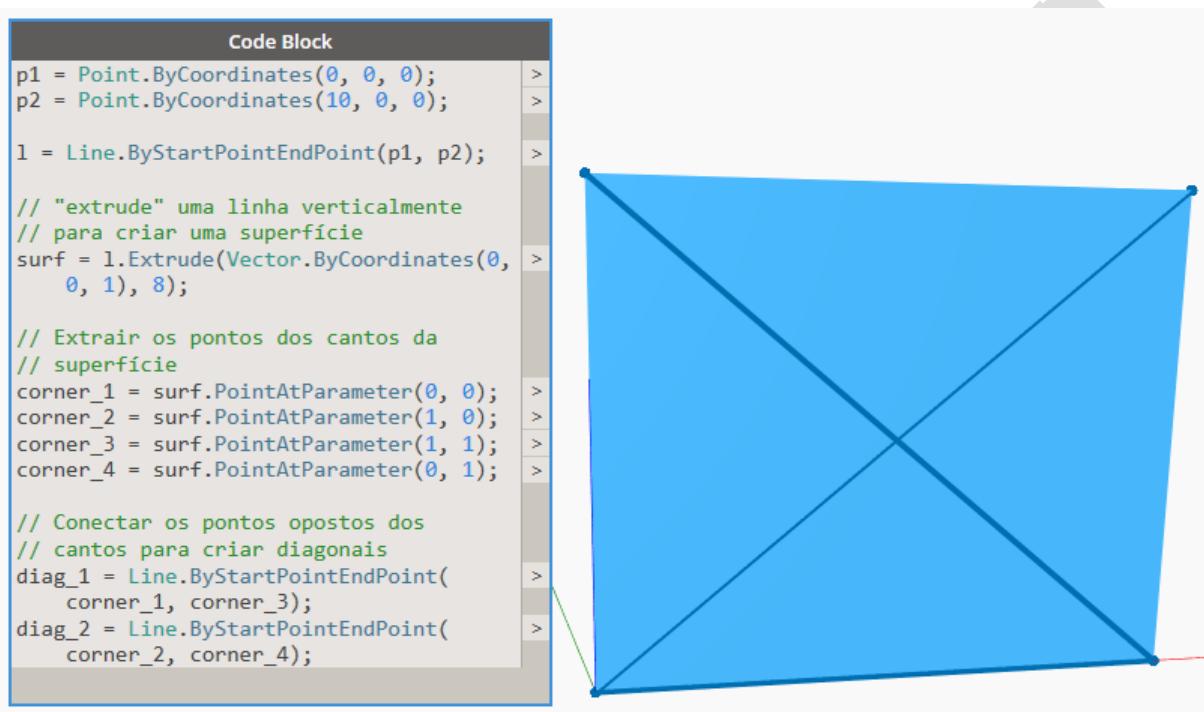


Funções

Quase toda a funcionalidade demonstrada no DesignScript até agora é expressa por meio de funções. Você pode dizer que um comando é uma função quando ele contém uma palavra-chave com o sufixo de parênteses contendo várias entradas. Quando uma função é chamada no DesignScript, uma grande quantidade de código é executada, processando as entradas e retornando um resultado. A função

construtora `Point.ByCoordinates (x: double, y: double, z: double)` recebe três entradas, as processa e retorna um objeto `Point`. Como a maioria das linguagens de programação, o DesignScript oferece aos programadores a capacidade de criar suas próprias funções. As funções são uma parte crucial dos scripts eficazes: o processo de pegar blocos de código com funcionalidade específica, envolvendo-os em uma descrição clara de entradas e saídas, adiciona legibilidade ao seu código e facilita a modificação e a reutilização.

Suponha que um programador tenha escrito um script para criar um suporte diagonal em uma superfície:



Este simples ato de criar diagonais sobre uma superfície, no entanto, requer várias linhas de código. Se quiséssemos encontrar as diagonais de centenas, senão milhares de superfícies, um sistema de extração individual de pontos de canto e desenho de diagonais seria completamente impraticável. Criar uma função para extrair as diagonais de uma superfície permite que um programador aplique a funcionalidade de várias linhas de código a qualquer número de entradas de base.

As funções são criadas escrevendo a palavra-chave `def`, seguida do nome da função e uma lista de entradas de função, chamadas argumentos, entre parênteses. O código que a função contém está entre chaves: `{}`. No DesignScript, as funções devem retornar um valor, indicado por "atribuir" um valor à variável de palavra-chave de retorno. Por exemplo (a caixa vermelha indica erro na sintaxe, porque é apenas para exemplificar):

```

Code Block
def functionName(argument1, arguent2, etc, etc, ...)
{
    // Código vai estar aqui
    return = returnVariable;
};

```

Essa função usa um único argumento e retorna esse argumento multiplicado por 2:

```

Code Block
def getTimesTwo(arg)
{
    return = arg * 2;
}

times_two = getTimesTwo(10); >

```

Watch
 > >
 20

As funções não precisam necessariamente receber argumentos. Uma função simples para retornar à proporção áurea é assim:

```

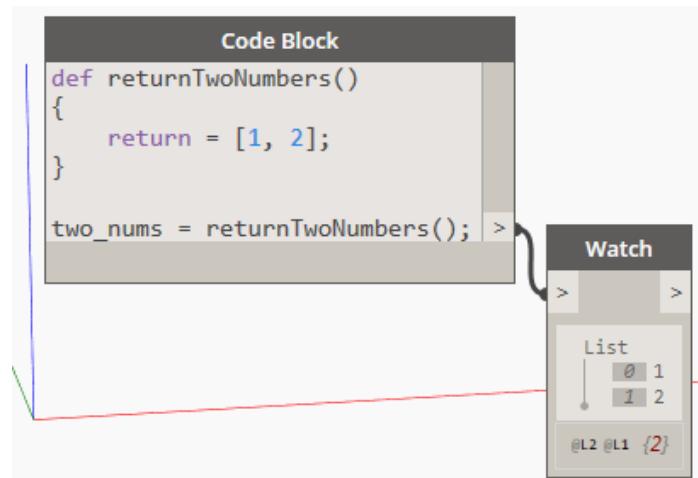
Code Block
def getGoldenRatio()
{
    return = 1.61803399;
}

gr = getGoldenRatio(); >

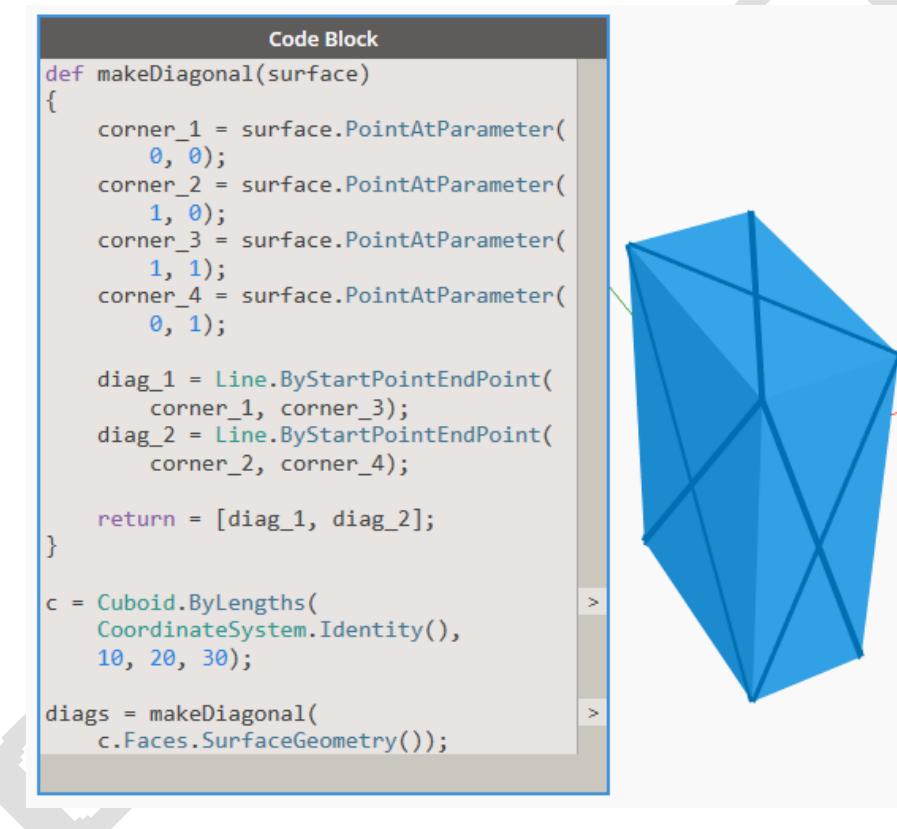
```

Watch
 > >
 1.61803399

Antes de criarmos uma função para quebrar nosso código diagonal, observe que as funções podem retornar apenas um único valor, mas nosso código diagonal gera duas linhas. Para contornar esse problema, podemos agrupar dois objetos entre colchetes [], criando um único objeto de coleção. Por exemplo, aqui está uma função simples que retorna dois valores:



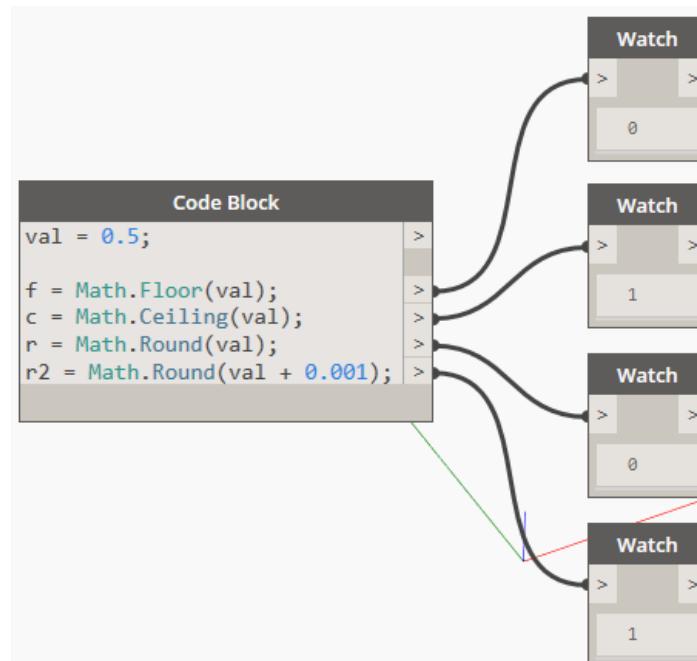
Se envolvemos o código diagonal em uma função, podemos criar diagonais sobre uma série de superfícies, por exemplo, as faces de um cuboide.



Matemática

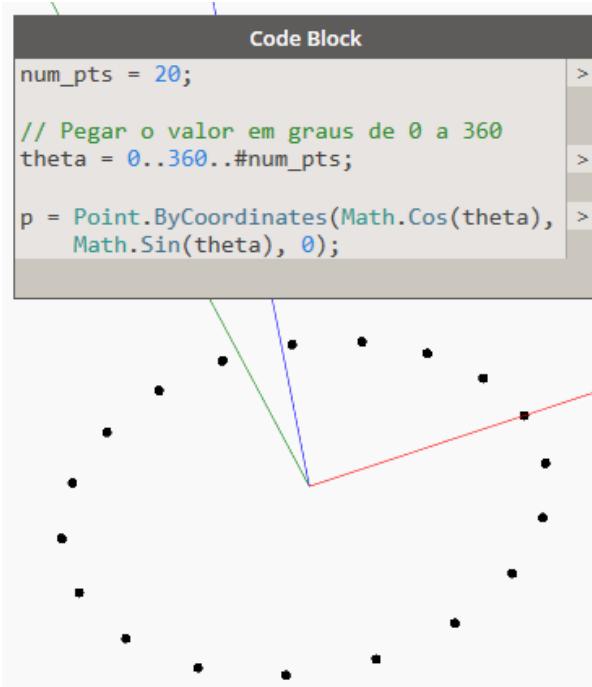
A biblioteca padrão do Dynamo contém uma variedade de funções matemáticas para ajudar a escrever algoritmos e manipular dados. As funções matemáticas são prefixadas com o *namespace Math*, exigindo que você acrescente funções com "Math" para usá-las.

As funções *Floor*, *Ceiling* e *Round* permitem converter entre números de ponto flutuante e números inteiros com resultados previsíveis. Todas as três funções recebem um único número de ponto flutuante como entrada, embora *Floor* retorne um número inteiro sempre arredondando para baixo, *Ceiling* retorne um número inteiro sempre arredondando para cima e *Round* para o número inteiro mais próximo.

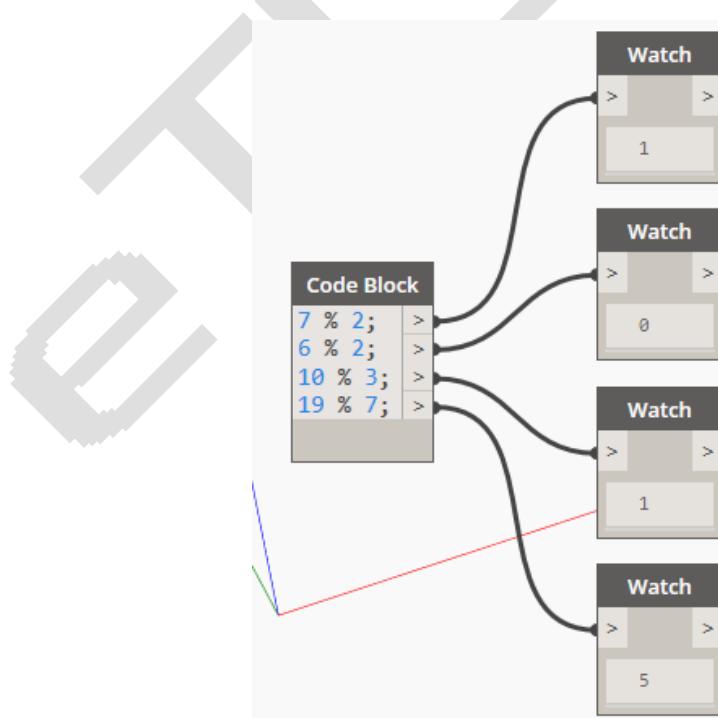


O Dynamo também contém um conjunto padrão de funções trigonométricas para calcular os ângulos seno, cosseno, tangente, arco-seno, arco-cosseno e arco-tangente, com as funções Sin, Cos, Tan, Asin, Acos e Atan, respectivamente.

Embora uma descrição abrangente da trigonometria esteja além do escopo deste e-book, as funções seno e cosseno ocorrem com frequência em projetos computacionais devido à sua capacidade de rastrear posições em um círculo com raio 1. Ao inserir um ângulo de grau crescente, geralmente rotulado como teta, em Cos para a posição x e Sin na posição y, as posições em um círculo são calculadas:



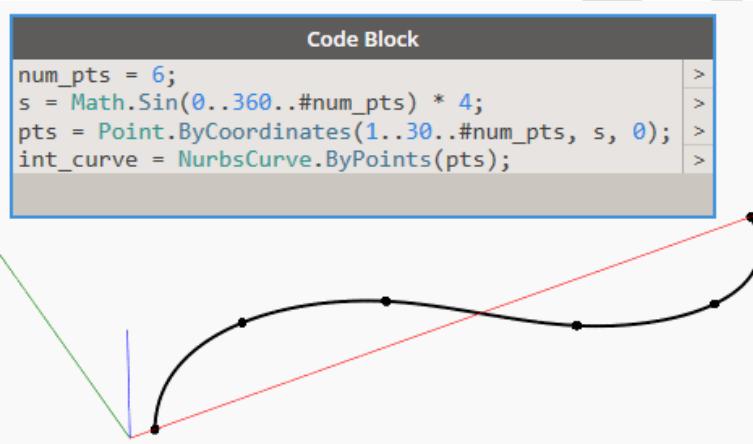
Um conceito de matemática relacionado que não faz parte estritamente da biblioteca padrão de *Math* é o operador de módulo. O operador do módulo, indicado por um sinal de porcentagem (%), retorna o restante de uma divisão entre dois números inteiros. Por exemplo, 7 dividido por 2 é 3 com 1 sobra (por exemplo, $2 \times 3 + 1 = 7$). O módulo entre 7 e 2, portanto, é 1. Por outro lado, 2 se divide uniformemente em 6 e, portanto, o módulo entre 6 e 2 é 0. O exemplo a seguir ilustra o resultado de várias operações de módulo.



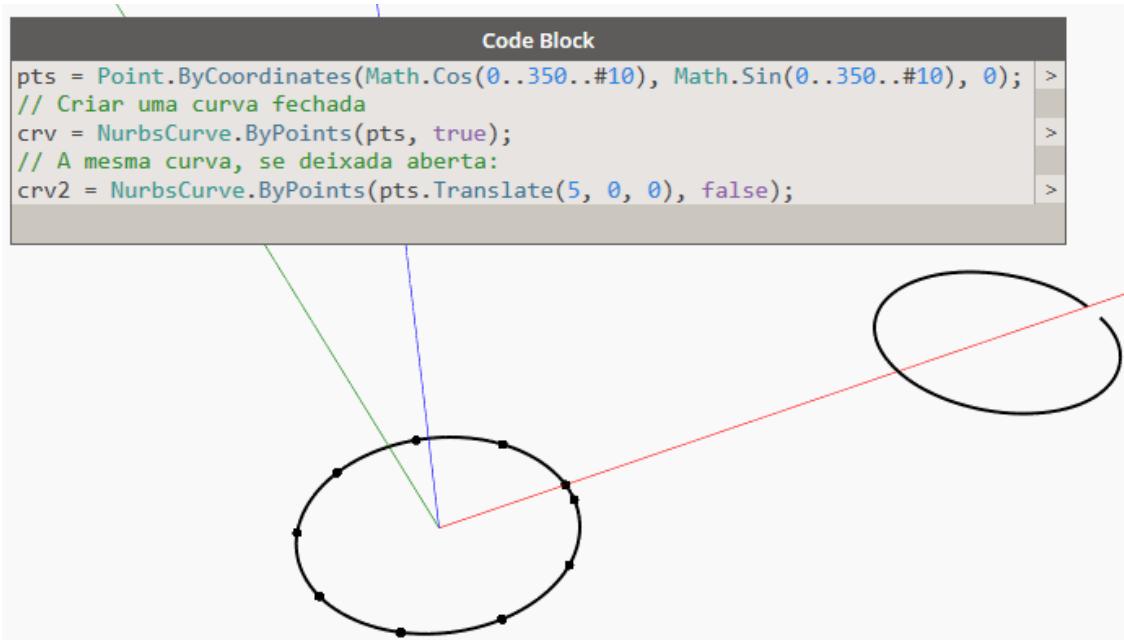
Curvas: Pontos Interpretados e de Controle

Existem duas maneiras fundamentais de criar curvas de forma livre no Dynamo: especificar uma coleção de pontos e fazer com que o Dynamo interprete uma curva suave entre eles, ou um método de nível mais baixo, especificando os pontos de controle subjacentes de uma curva de um certo grau. Curvas interpretadas são úteis quando um designer sabe exatamente a forma que uma linha deve assumir ou se o design possui restrições específicas para onde a curva pode e não pode passar. As curvas especificadas por meio de pontos de controle são essencialmente uma série de segmentos de linha reta que um algoritmo suaviza em uma forma final de curva. A especificação de uma curva por meio de pontos de controle pode ser útil para explorações de formas de curva com graus variados de suavização ou quando é necessária uma continuidade suave entre os segmentos da curva.

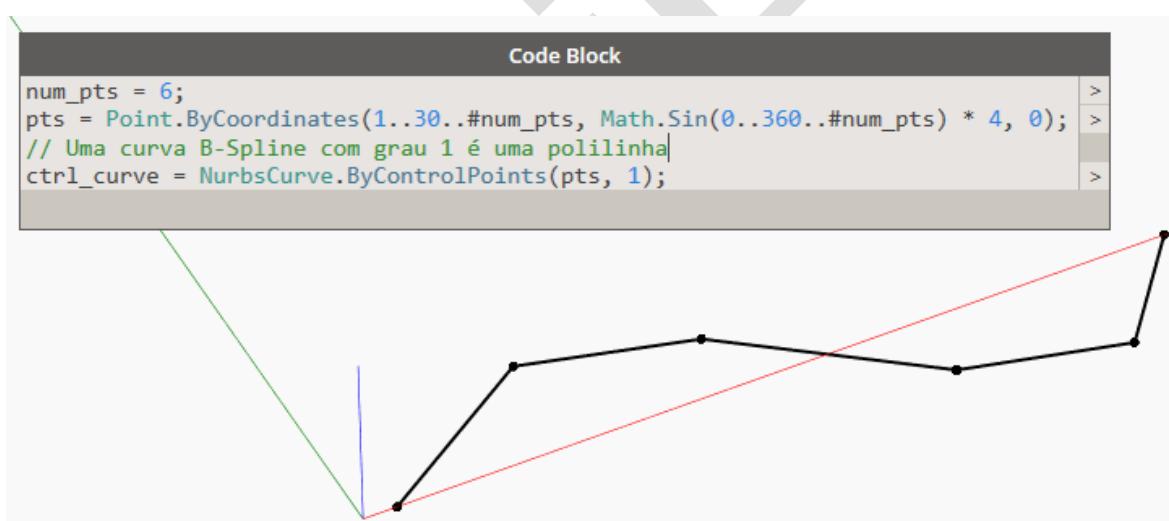
Para criar uma curva interpretada, basta passar uma coleção de *Points* para o método *NurbsCurve.ByPoints*.



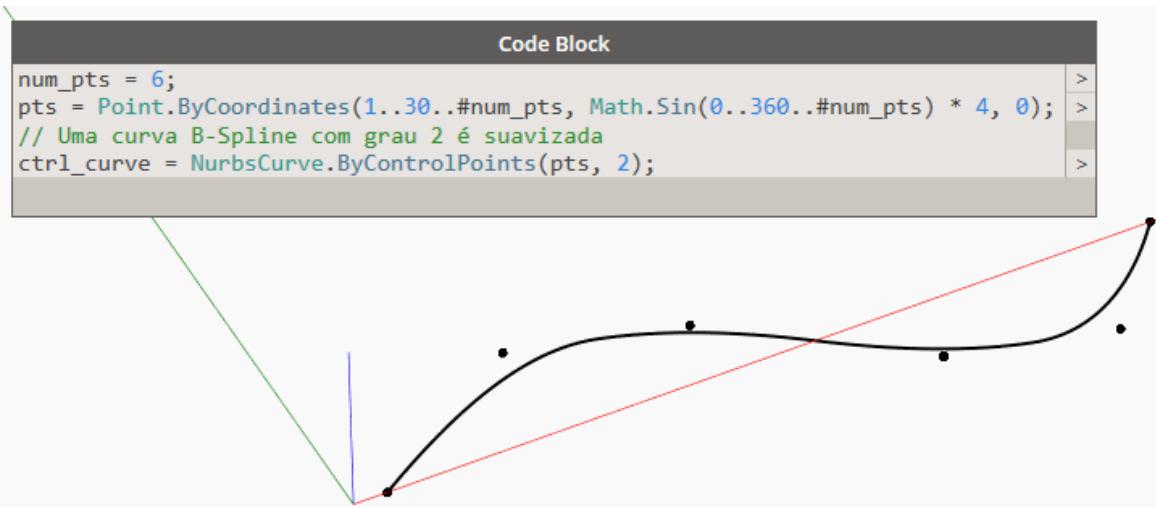
A curva gerada intercepta cada um dos pontos de entrada, começando e terminando no primeiro e no último ponto da coleção, respectivamente. Um parâmetro periódico opcional pode ser usado para criar uma curva periódica fechada. O Dynamo preencherá automaticamente o segmento ausente, portanto, um ponto final duplicado (idêntico ao ponto inicial) não é necessário.



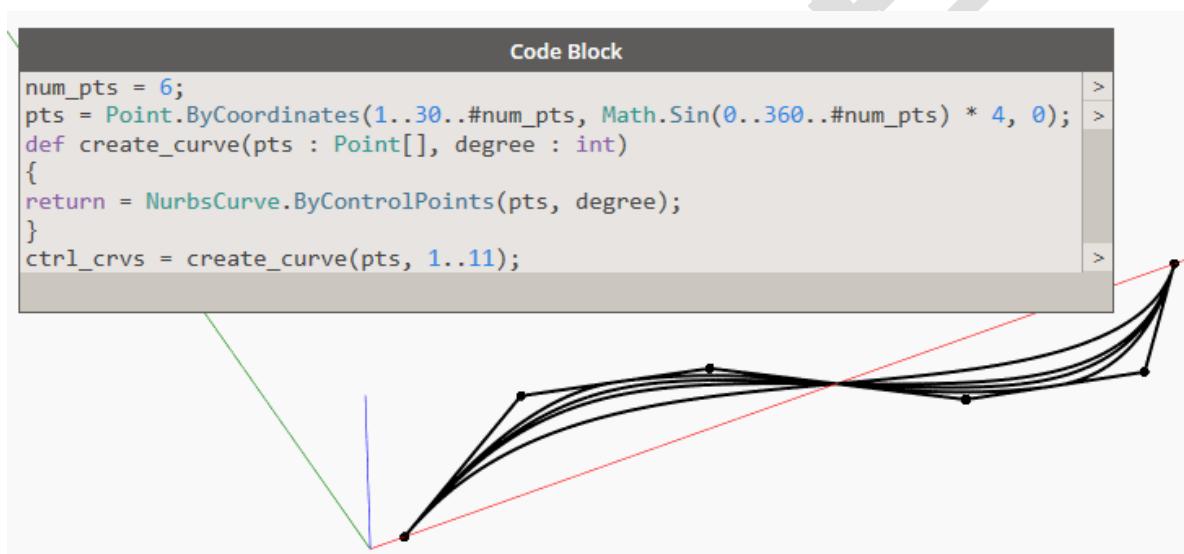
NurbsCurves são gerados da mesma maneira, com os pontos de entrada representando os pontos finais de um segmento de linha reta, e um segundo parâmetro que especifica a quantidade e o tipo de suavização pela qual a curva passa, chamada de grau. Uma curva com grau 1 não possui suavização; é uma polilinha.



Uma curva com grau 2 é suavizada de modo que a curva se cruze e seja tangente ao ponto médio dos segmentos da polilinha:

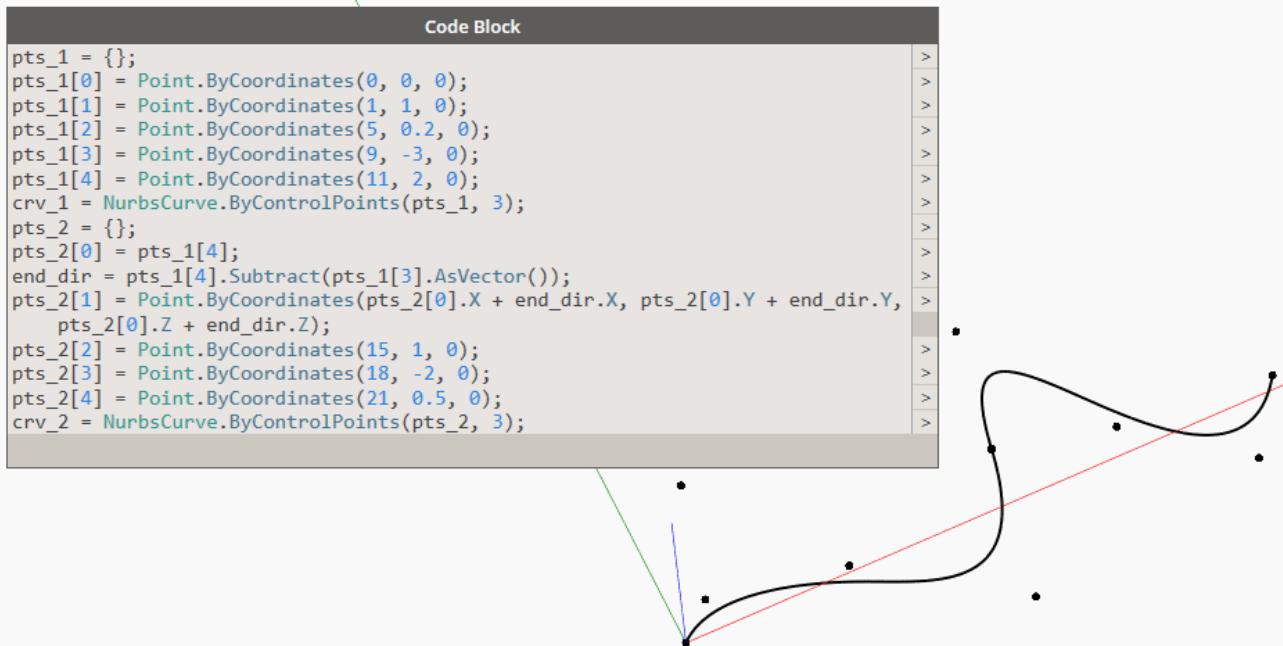


O Dynamo suporta curvas NURBS (*spline* B racional não uniforme) até o grau 20, e o script a seguir ilustra o efeito que os níveis crescentes de suavização têm no formato de uma curva:



Observe que você deve ter pelo menos mais um ponto de controle que o grau da curva.

Outro benefício da construção de curvas pelos vértices de controle é a capacidade de manter a tangência entre segmentos de curvas individuais. Isso é feito extraíndo a direção entre os dois últimos pontos de controle e continuando essa direção com os dois primeiros pontos de controle da curva a seguir. O exemplo a seguir cria duas curvas NURBS separadas que, no entanto, são tão suaves quanto uma curva:

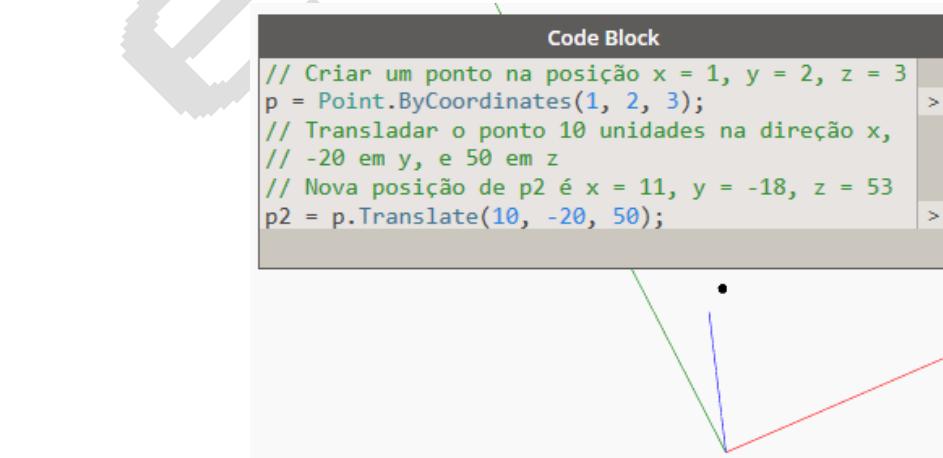


Esta é uma descrição muito simplificada da geometria da curva NURBS, para uma discussão mais precisa e detalhada, veja Pottmann, et al, 2007.

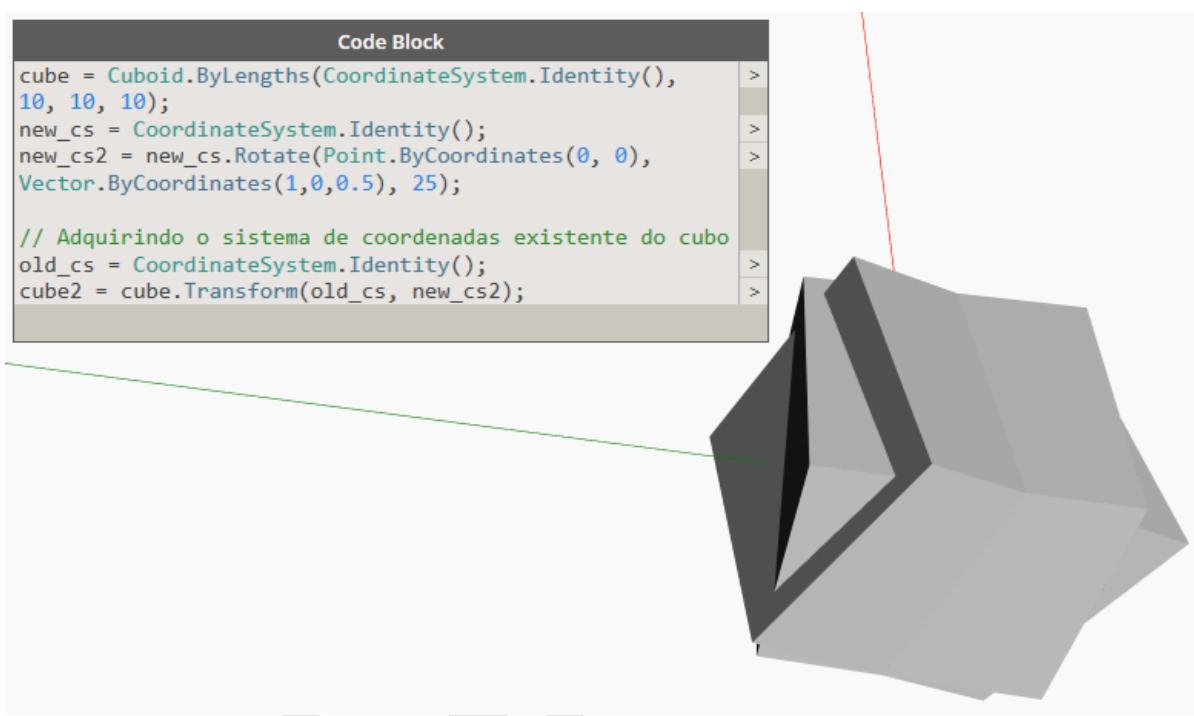
Translação, Rotação e Outras Transformações

Certos objetos de geometria podem ser criados declarando explicitamente as coordenadas x, y e z no espaço tridimensional. Mais frequentemente, no entanto, a geometria é movida para sua posição final usando transformações geométricas no próprio objeto ou no sistema de coordenadas subjacente.

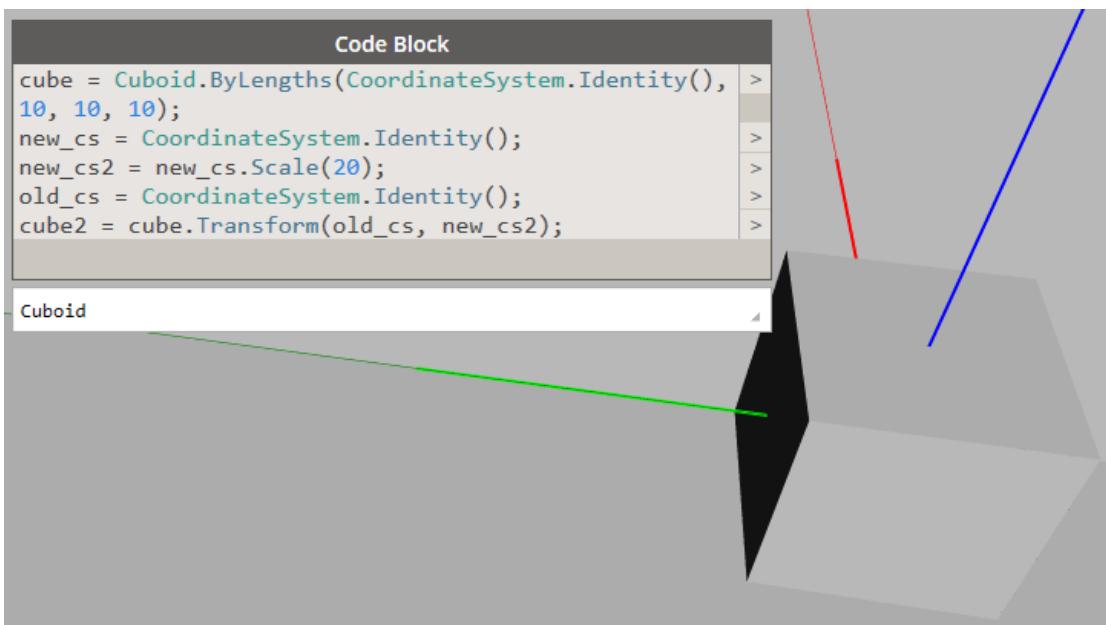
A transformação geométrica mais simples é uma translação, que move um objeto para um número especificado de unidades nas direções x, y e z.



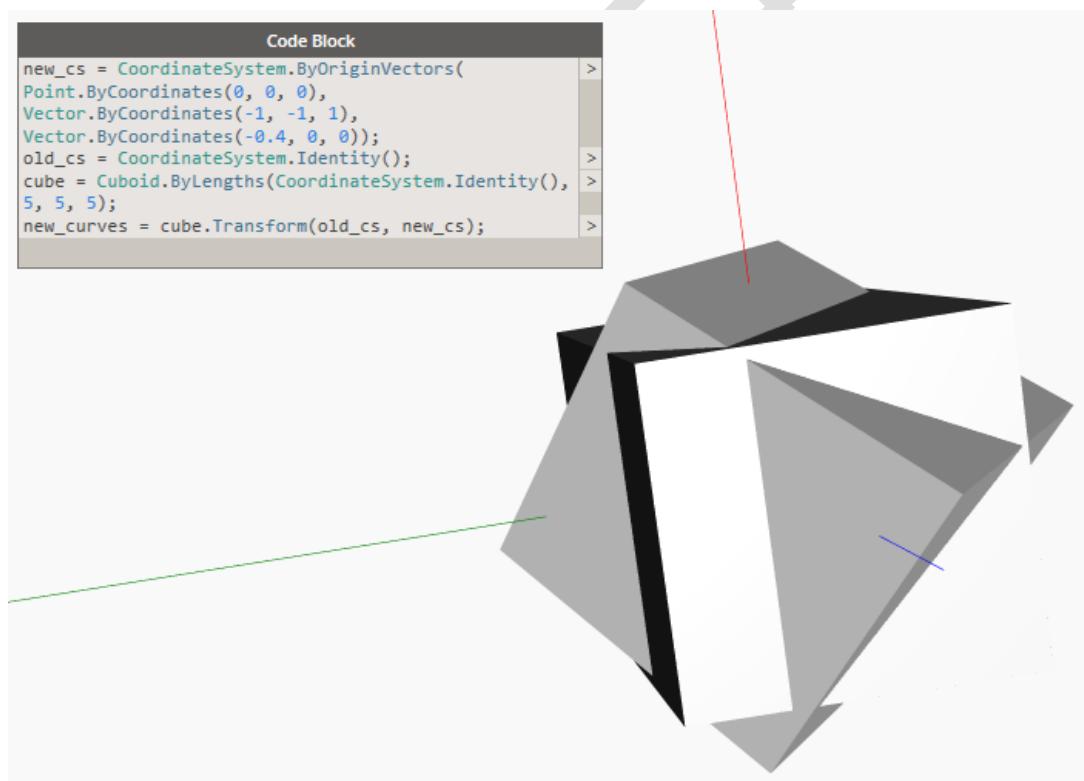
Embora todos os objetos no Dynamo possam ser traduzidos anexando o método *.Translate* ao final do nome do objeto, transformações mais complexas exigem a transformação do objeto de um sistema CoordinateSystem subjacente em um novo sistema CoordinateSystem. Por exemplo, para rotacionar um objeto 45 graus ao redor do eixo x, transformaríamos o objeto do seu CoordinateSystem existente sem rotação, para um CoordinateSystem que tivesse sido rotacionado 45 graus ao redor do eixo x com o método *.Transform*



Além de serem traduzidos e rotacionados, o CoordinateSystems também pode ser criado em escala ou cisalhamento. Um CoordinateSystem pode ser dimensionado com o método *.Scale*.



CoordinateSystems cisalhados são criados inserindo vetores não ortogonais no construtor CoordinateSystem.



Escala e cisalhamento são transformações geométricas comparativamente mais complexas do que rotação e translação; portanto, nem todo objeto Dynamo pode sofrer essas transformações. A tabela

a seguir descreve quais objetos do Dynamo podem ter o CoordinateSystems dimensionado de maneira não uniforme e o CoordinateSystems cisalhados.

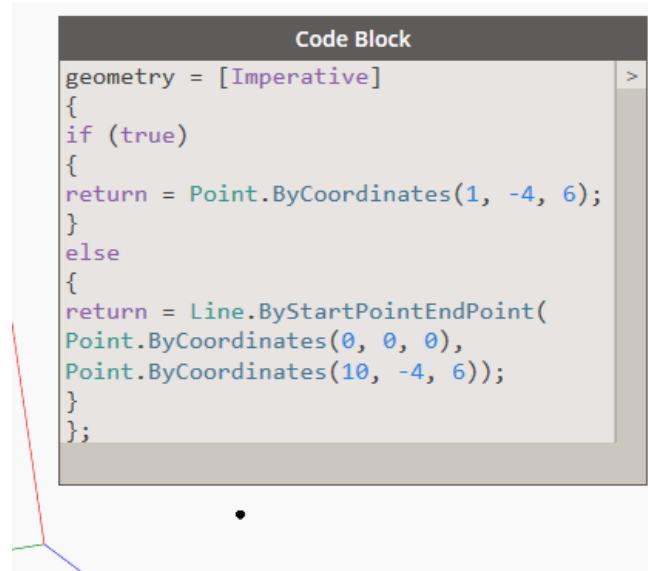
Classe	CoordinateSystem não uniformemente escalonado	CoordinateSystem cisalhado
Arco	Não	Não
NurbsCurve	Sim	Sim
NurbsSurface	Não	Não
Circle	Não	Não
Line	Sim	Sim
Plane	Não	Não
Point	Sim	Sim
Polygon	Não	Não
Solid	Não	Não
Surface	Não	Não
Text	Não	Não

Condicionais e Lógica Booleana

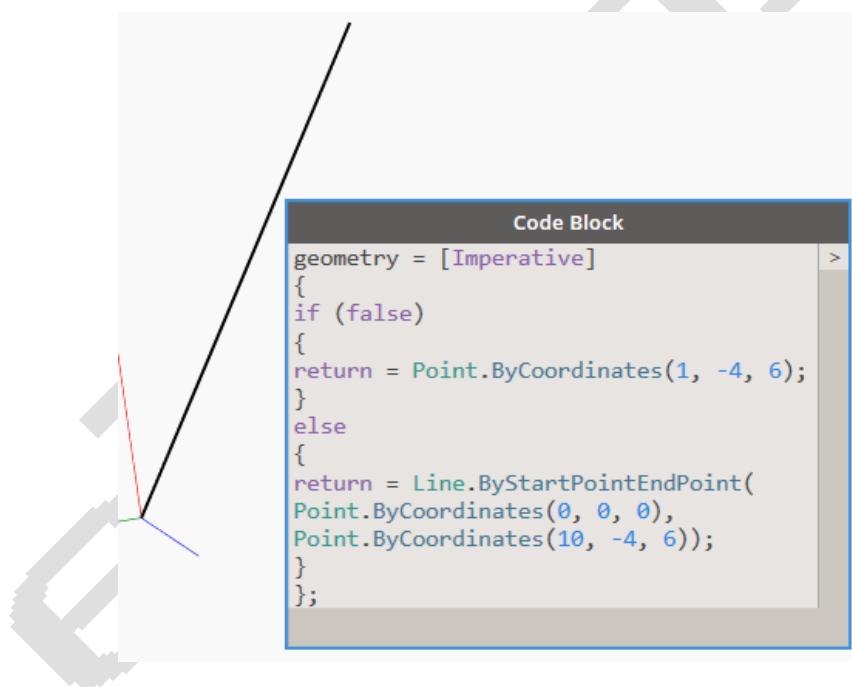
Um dos recursos mais poderosos de uma linguagem de programação é a capacidade de observar os objetos existentes em um programa e variar a execução do programa de acordo com as qualidades desses objetos. As linguagens de programação mediam entre exames das qualidades de um objeto e a execução de código específico por meio de um sistema chamado lógica booleana.

A lógica booleana examina se as declarações são verdadeiras ou falsas. Toda declaração na lógica booleana será verdadeira ou falsa, não há outros estados; não, talvez, possível ou talvez exista. A maneira mais simples de indicar que uma instrução booleana é verdadeira é com a palavra-chave *true*. Da mesma forma, a maneira mais simples de indicar que uma declaração é falsa é com a palavra-chave *false*. A instrução *if* permite determinar se uma instrução é verdadeira ou falsa: se for verdadeira, a primeira parte do bloco de código será executada; se for falsa, o segundo bloco de código será executado.

No exemplo a seguir, a instrução *if* contém uma instrução booleana verdadeira, para que o primeiro bloco seja executado e um *Point* seja gerado:



Se a instrução contida for alterada para *false*, o segundo bloco de código será executado e uma linha será gerada:

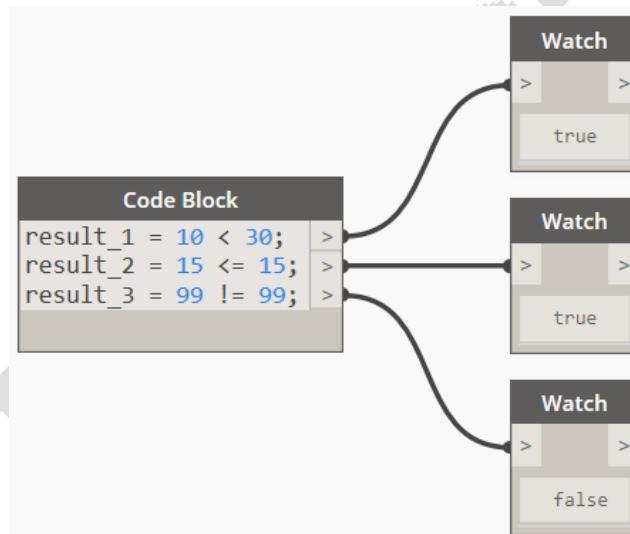


Declarações booleanas estáticas como essas não são particularmente úteis; o poder da lógica booleana vem do exame das qualidades dos objetos em seu script. A lógica booleana possui seis operações básicas para avaliar valores: menor que (<), maior que (>), menor que ou igual (<=), maior que ou igual (>=), igual (==) e diferente (!=). O gráfico a seguir descreve os resultados booleanos.

<	Retorna true se o número no lado esquerdo for menor que o número no lado direito.
>	Retorna true se o número no lado esquerdo for maior que o número no lado direito.
<=	Retorna verdadeiro do número no lado esquerdo é menor ou igual ao número no lado direito. *
> =	Retorna verdadeiro do número no lado esquerdo é maior ou igual ao número no lado direito. *
==	Retorna true se os dois números forem iguais *
!=	Retorna true se os dois números não forem iguais *

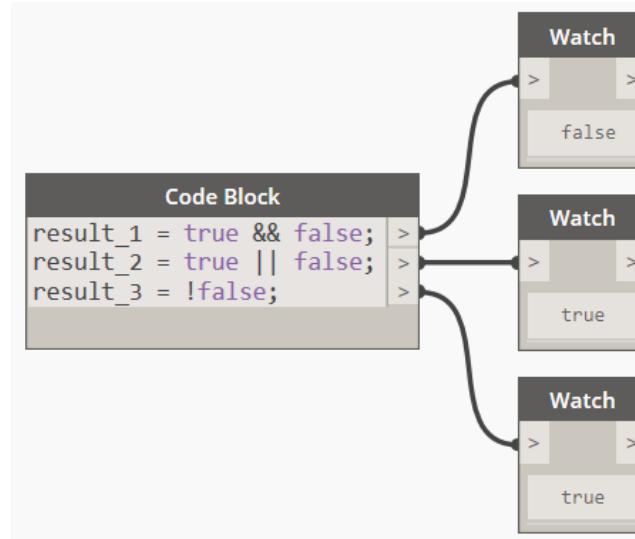
* consulte o tema "Tipos de números" para saber as limitações do teste de igualdade entre dois números de ponto flutuante.

O uso de um desses seis operadores em dois números retorna *true* ou *false*.

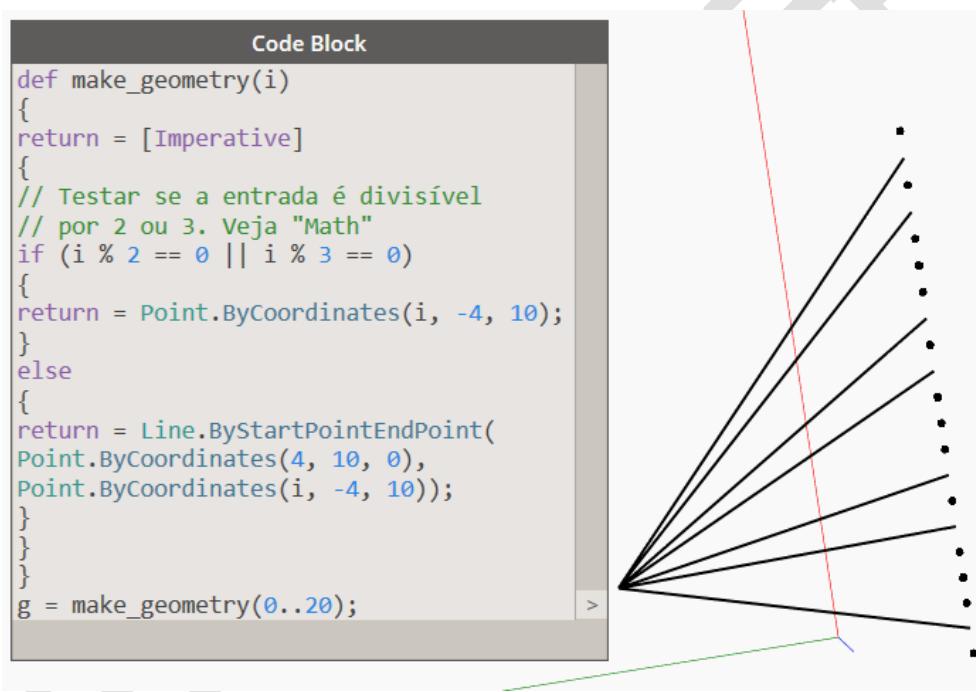


Existem outros três operadores booleanos para comparar declarações verdadeiras e falsas: e (&&), ou (||) e não (!).

&&	Retorna <i>true</i> se os valores de ambos os lados forem verdadeiros.
	Retorna <i>true</i> se um dos valores de ambos os lados for verdadeiro.
!	Retorna o oposto booleano.



A refatoração do código no exemplo original demonstra diferentes caminhos de execução de código com base nas entradas alteradas de uma expressão de intervalo:



Looping

Loops são comandos para repetir a execução em um bloco de código. O número de vezes que um loop é chamado pode ser controlado por uma coleção, em que um loop é chamado com cada elemento da coleção como entrada ou com uma expressão booleana, em que o loop é chamado até que a expressão booleana retorne *false*. Os loops podem ser usados para gerar coleções, procurar uma solução ou adicionar repetição sem expressões de intervalo.

A instrução *while* avalia uma expressão booleana e continua reexecutando o bloco de código contido até que a expressão booleana seja *true*. Por exemplo, esse script cria e recria continuamente uma linha até ter um comprimento maior que 10:

```

Code Block
geometry = [Imperative]
{
x = 1;
start = Point.ByCoordinates(0, 0, 0);
end = Point.ByCoordinates(x, x, x);
line = Line.ByStartPointEndPoint(start, end);
while (line.Length < 10)
{
x = x + 1;
end = Point.ByCoordinates(x, x, x);
line = Line.ByStartPointEndPoint(start, end);
}
return = line;
};

Line

```

No código Dynamo associativo, se uma coleção de elementos for usada como entrada para uma função que normalmente assume um valor único, a função será chamada individualmente para cada membro de uma coleção. No código imperativo do Dynamo, um programador tem a opção de escrever código que itera manualmente sobre a coleção, extraíndo membros individuais da coleção, um de cada vez.

A instrução *for* extrai elementos de uma coleção em uma variável nomeada, uma vez para cada membro de uma coleção. A sintaxe para *for* é: *for* ("variável extraída" *in* "coleção de entrada").

```

Code Block
geometry = [Imperative]
{
collection = 0..10;
points = {};
for (i in collection)
{
points[i] = Point.ByCoordinates(i, 0, 0);
}
return = points;
};

List
0 Point(X = 0.000, Y = 0.000, Z = 0.000)
1 Point(X = 1.000, Y = 0.000, Z = 0.000)
2 Point(X = 2.000, Y = 0.000, Z = 0.000)
3 Point(X = 3.000, Y = 0.000, Z = 0.000)
4 Point(X = 4.000, Y = 0.000, Z = 0.000)
5 Point(X = 5.000, Y = 0.000, Z = 0.000)
6 Point(X = 6.000, Y = 0.000, Z = 0.000)
7 Point(X = 7.000, Y = 0.000, Z = 0.000)
8 Point(X = 8.000, Y = 0.000, Z = 0.000)
9 Point(X = 9.000, Y = 0.000, Z = 0.000)
10 Point(X = 10.000, Y = 0.000, Z = 0.000)

@L2 @L1 {11}

```

Guias de Replicação

A linguagem Dynamo foi criada como uma ferramenta específica de domínio para arquitetos, designers e engenheiros e, como tal, possui vários recursos de linguagem especificamente adaptados para essas disciplinas. Um elemento comum nessas disciplinas é a prevalência de objetos dispostos em redes repetitivas, desde paredes de tijolos e pisos de azulejos a painéis de fachadas e grades de colunas. Embora as expressões de intervalo ofereçam um meio conveniente de gerar coleções unidimensionais de elementos, os guias de replicação oferecem um meio conveniente de gerar coleções bidimensionais e tridimensionais.

Os guias de replicação usam duas ou três coleções unidimensionais e emparelham os elementos para gerar uma coleção bidimensional ou tridimensional. As guias de replicação são indicadas colocando os símbolos <1>, <2> ou <3> após duas ou três coleções em uma única linha de código. Por exemplo, podemos usar expressões de intervalo para gerar duas coleções unidimensionais e usá-las para gerar uma coleção de pontos:

The screenshot shows the Dynamo interface with a 'Code Block' panel at the top containing the following script:

```
Code Block
x_vals = 0..10;
y_vals = 0..10;
p = Point.ByCoordinates(x_vals, y_vals, 0);
```

Below it is a 'List' panel displaying a series of 11 points:

Index	Point
0	Point(X = 0.000, Y = 0.000, Z = 0.000)
1	Point(X = 1.000, Y = 1.000, Z = 0.000)
2	Point(X = 2.000, Y = 2.000, Z = 0.000)
3	Point(X = 3.000, Y = 3.000, Z = 0.000)
4	Point(X = 4.000, Y = 4.000, Z = 0.000)
5	Point(X = 5.000, Y = 5.000, Z = 0.000)
6	Point(X = 6.000, Y = 6.000, Z = 0.000)
7	Point(X = 7.000, Y = 7.000, Z = 0.000)
8	Point(X = 8.000, Y = 8.000, Z = 0.000)
9	Point(X = 9.000, Y = 9.000, Z = 0.000)
10	Point(X = 10.000, Y = 10.000, Z = 0.000)

At the bottom of the list panel, there are buttons for '@L2' and '@L1' and a count of '{11}'.

Neste exemplo, o primeiro elemento de `x_vals` é emparelhado com o primeiro elemento de `y_vals`, o segundo com o segundo e assim por diante durante todo o comprimento da coleção. Isso gera pontos com valores (0, 0, 0), (1, 1, 0), (2, 2, 0), (3, 3, 0), etc. Se aplicarmos um guia de replicação a essa mesma linha de código, o Dynamo poderá gerar uma grade bidimensional a partir das duas coleções unidimensionais:

Code Block

```
x_vals = 0..10;
y_vals = 0..10;
// Aplicando guias de replicação para as duas coleções
p = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);
```

List

- 0 List
 - 0 Point(X = 0.000, Y = 0.000, Z = 0.000)
 - 1 Point(X = 0.000, Y = 1.000, Z = 0.000)
 - 2 Point(X = 0.000, Y = 2.000, Z = 0.000)
 - 3 Point(X = 0.000, Y = 3.000, Z = 0.000)
 - 4 Point(X = 0.000, Y = 4.000, Z = 0.000)
 - 5 Point(X = 0.000, Y = 5.000, Z = 0.000)
 - 6 Point(X = 0.000, Y = 6.000, Z = 0.000)
 - 7 Point(X = 0.000, Y = 7.000, Z = 0.000)
 - 8 Point(X = 0.000, Y = 8.000, Z = 0.000)
 - 9 Point(X = 0.000, Y = 9.000, Z = 0.000)
 - 10 Point(X = 0.000, Y = 10.000, Z = 0.000)
- 1 List
 - 0 Point(X = 1.000, Y = 0.000, Z = 0.000)
 - 1 Point(X = 1.000, Y = 1.000, Z = 0.000)
 - 2 Point(X = 1.000, Y = 2.000, Z = 0.000)

③ L3 ② L2 ① L1 {121}

Ao aplicar guias de replicação a `x_vals` e `y_vals`, o Dynamo gera todas as combinações possíveis de valores entre as duas coleções, emparelhando primeiro o primeiro elemento `x_vals` com todos os elementos em `y_vals` e, em seguida, emparelhando o segundo elemento de `x_vals` com todos os elementos em `y_vals`, e assim para cada elemento de `x_vals`.

A ordem dos números do guia de replicação (`<1>`, `<2>` e / ou `<3>`) determina a ordem da coleção subjacente. No exemplo a seguir, as mesmas duas coleções unidimensionais são usadas para formar duas coleções bidimensionais, embora com a ordem de `<1>` e `<2>` trocadas.

Code Block

```
x_vals = 0..10;
y_vals = 0..10;
p1 = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);
// Aplicando guias de replicação com a ordem trocada
// e definindo os pontos 14 unidades acima
p2 = Point.ByCoordinates(x_vals<2>, y_vals<1>, 14);
curve1 = NurbsCurve.ByPoints(Flatten(p1));
curve2 = NurbsCurve.ByPoints(Flatten(p2));
```

NurbsCurve(Degree = 3)

curve1 e curve2 traçam a ordem gerada dos elementos nas duas matrizes; observe que eles são girados 90 graus um para o outro. p1 foi criado extraíndo elementos de x_vals e emparelhando-os com y_vals, enquanto p2 foi criado extraíndo elementos de y_vals e emparelhando-os com x_vals.

Os guias de replicação também funcionam em três dimensões, emparelhando uma terceira coleção com um terceiro símbolo de replicação, <3>.

```
Code Block
x_vals = 0..10;
y_vals = 0..10;
z_vals = 0..10;
// Gera uma matriz 3D de pontos
p = Point.ByCoordinates(x_vals<1>,y_vals<2>,z_vals<3>);
```

```
List
  0 List
    0 List
      0 Point(X = 0.000, Y = 0.000, Z = 0.000)
      1 Point(X = 0.000, Y = 0.000, Z = 1.000)
      2 Point(X = 0.000, Y = 0.000, Z = 2.000)
      3 Point(X = 0.000, Y = 0.000, Z = 3.000)
      4 Point(X = 0.000, Y = 0.000, Z = 4.000)
      5 Point(X = 0.000, Y = 0.000, Z = 5.000)
      6 Point(X = 0.000, Y = 0.000, Z = 6.000)
      7 Point(X = 0.000, Y = 0.000, Z = 7.000)
      8 Point(X = 0.000, Y = 0.000, Z = 8.000)
      9 Point(X = 0.000, Y = 0.000, Z = 9.000)
      10 Point(X = 0.000, Y = 0.000, Z = 10.000)
    1 List
      0 Point(X = 0.000, Y = 1.000, Z = 0.000)
      1 Point(X = 0.000, Y = 1.000, Z = 1.000)
```

@L4 @L3 @L2 @L1 {1331}

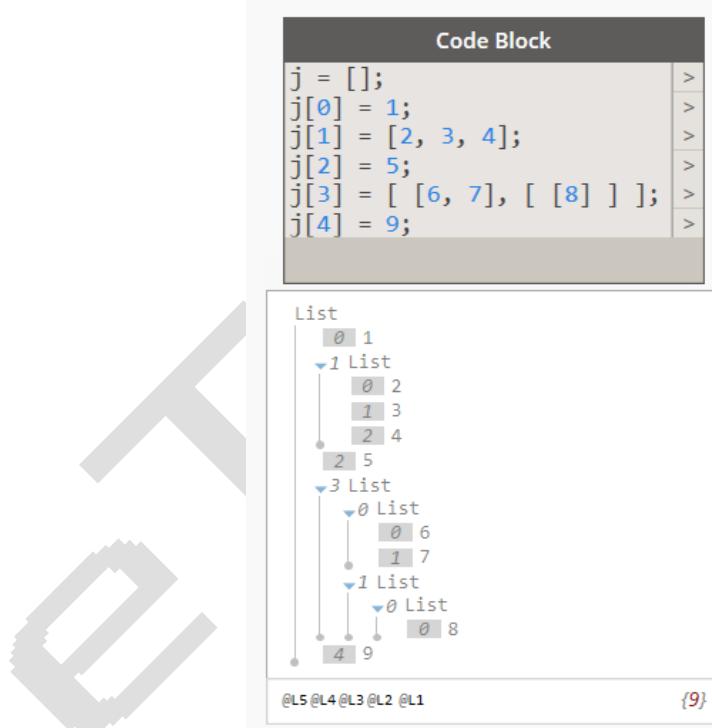
Isso gera todas as combinações possíveis de valores, combinando os elementos de x_vals, y_vals e z_vals.

Coleções Classificadas e Coleções Irregulares

A classificação de uma coleção é definida como a maior profundidade de elementos dentro de uma coleção. Uma coleção de valores únicos tem uma classificação de 1, enquanto uma coleção de coleções de valores únicos tem uma classificação de 2. A classificação pode ser definida livremente como o número de operadores de colchetes ([]) necessários para acessar o membro mais profundo de uma coleção. As coleções da classificação 1 precisam apenas de um colchete quadrado para acessar o membro mais profundo, enquanto as coleções da classificação 3 requerem três colchetes subsequentes. A tabela a seguir descreve coleções das classificações 1-3, embora as coleções possam existir até qualquer profundidade de classificação.

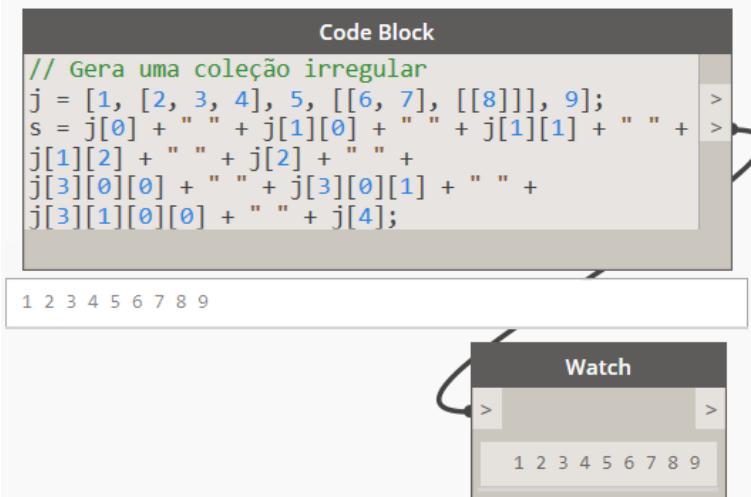
Classificação	Coleção	Acessar o 1º elemento
1	[1, 2, 3, 4, 5]	collection[0]
2	[[1, 2], [3, 4], [5, 6]]	collection[0][0]
3	[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]	collection[0][0][0]
...		

Coleções com classificação mais alta geradas por expressões de intervalo e guias de replicação são sempre homogêneas, ou seja, todos os objetos de uma coleção estão na mesma profundidade (é acessado com o mesmo número de operadores []). No entanto, nem todas as coleções do Dynamo contêm elementos na mesma profundidade. Essas coleções são chamadas de irregulares, após o fato de que a profundidade aumenta para cima e para baixo ao longo do comprimento da coleção. O código a seguir gera uma coleção irregular:



Falha ao tentar executar operações não suportadas em uma coleção.

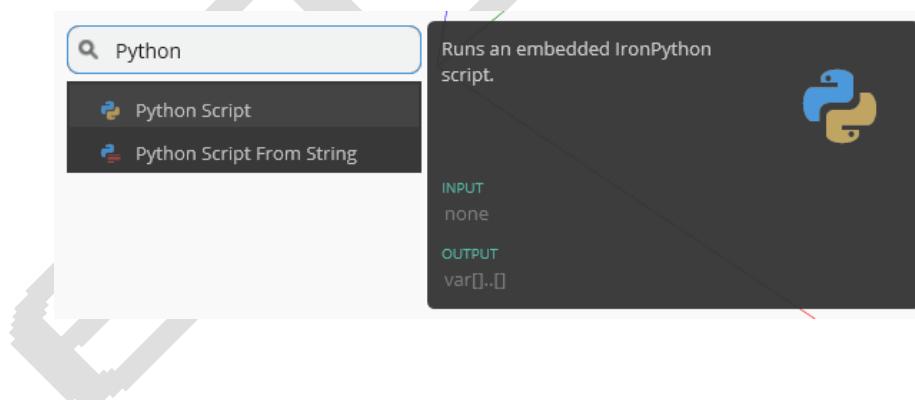
O exemplo a seguir mostra como acessar todos os elementos dessa coleção irregular:

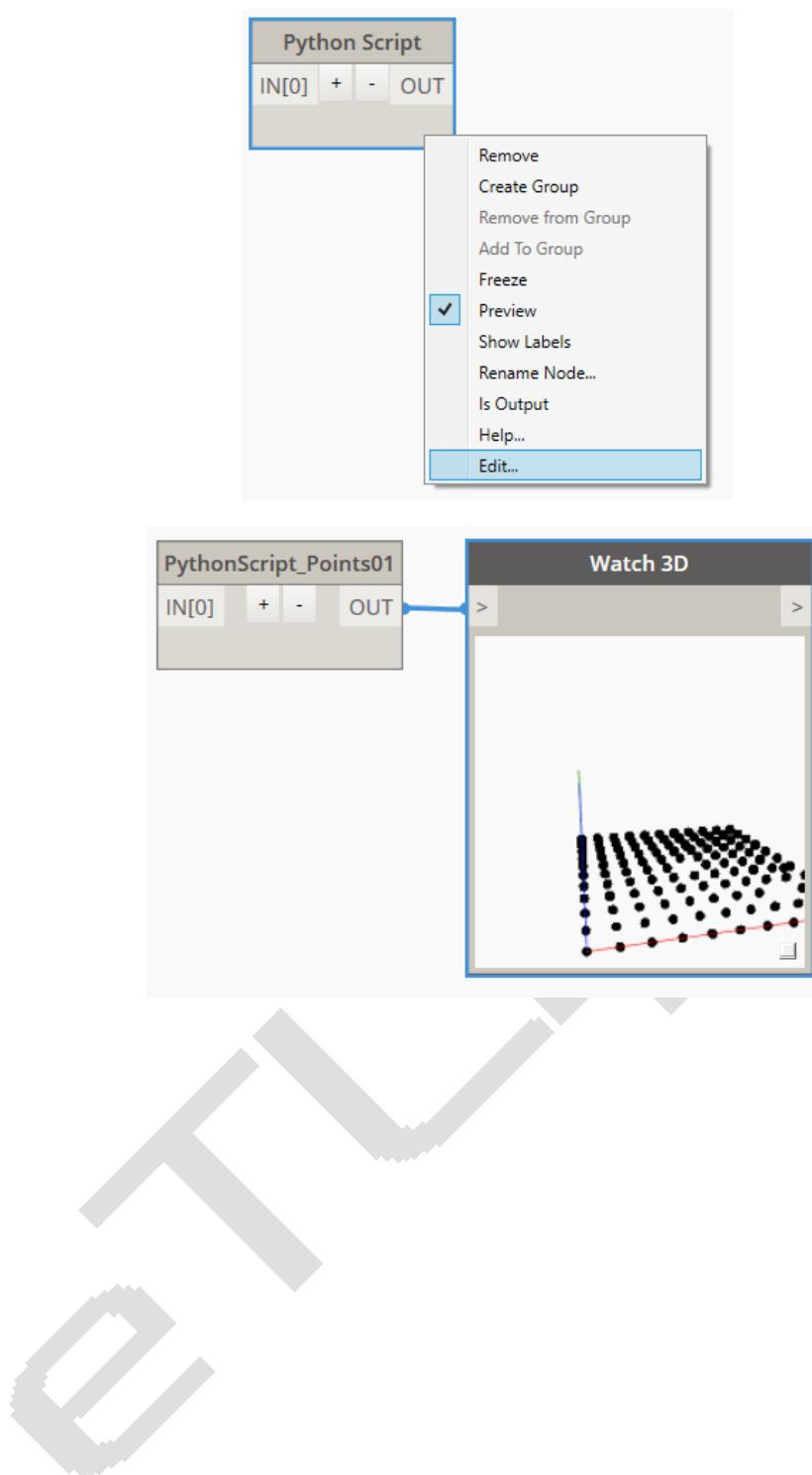


Geração de Elementos com Python

É possível a utilização do Python para interação com demais elementos no Dynamo. Esta funcionalidade permite, por exemplo, a geração de elementos, como pontos, como alguns scripts a seguir apresentados, e que serão usados em alguns exemplos na sequência deste estudo.

Os seguintes scripts Python geram matrizes de pontos para vários exemplos. Eles devem ser colados em um nó de script Python e em seguida editados.





PythonScript_Points01

```

1 # Load the Python Standard and DesignScript Libraries
2 import sys
3 import clr
4 clr.AddReference('ProtoGeometry')
5 from Autodesk.DesignScript.Geometry import *
6
7 # The inputs to this node will be stored as a list in the IN variables.
8 dataEnteringNode = IN
9
10 out_points = []
11
12 for i in range(11):
13     sub_points = []
14     for j in range(11):
15         z = 0
16         if (i == 5 and j == 5):
17             z = 1
18         elif (i == 8 and j == 2):
19             z = 1
20             sub_points.Add(Point.ByCoordinates(i, j, z))
21     out_points.Add(sub_points)
22
23 OUT = out_points

```

Run Save Changes Revert

PythonScript_Points01

```

# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

out_points = []

for i in range(11):
    sub_points = []

```

```

for j in      (11):
    z = 0
    if (i == 5 and j == 5):
        z = 1
    elif (i == 8 and j == 2):
        z = 1
    sub_points.    (Point.          (i, j, z))
    out_points.   (sub_points)

OUT = out_points

```

PythonScript_Points02

```

# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.           ('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

out_points = []

for i in range(11):
    z = 0
    if (i == 2):
        z = 1
    out_points.    (Point.          (i, 0, z))

OUT = out_points

```

PythonScript_Points03

```
# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.          (ProtoGeometry)
from Autodesk.DesignScript.Geometry import *
```

The inputs to this node will be stored as a list in the IN variables.

```
dataEnteringNode = IN
```

```
out_points = []
```

```
for i in      (11):
```

```
    z = 0
```

```
    if (i == 7):
```

```
        z = -1
```

```
        out_points.  (Point.  (Coordinate.  (i, 5, z)))
```

```
OUT = out_points
```

PythonScript_Points04

```
# Load the Python Standard and DesignScript Libraries
```

```
import sys
```

```
import clr
```

```
clr.          (ProtoGeometry)
```

```
from Autodesk.DesignScript.Geometry import *
```

The inputs to this node will be stored as a list in the IN variables.

```
dataEnteringNode = IN
```

```

out_points = []

for i in range(11):
    z = 0
    if (i == 5):
        z = 1
    out_points.append(Point(i, 10, z))

OUT = out_points

```

PythonScript_Points05

```

# Load the Python Standard and DesignScript Libraries
import sys
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *

# The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN

out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 1 and j == 1):
            z = 2
        elif (i == 8 and j == 1):
            z = 2
        sub_points.append(Point(i, 10, z))
    out_points.append(sub_points)

OUT = out_points

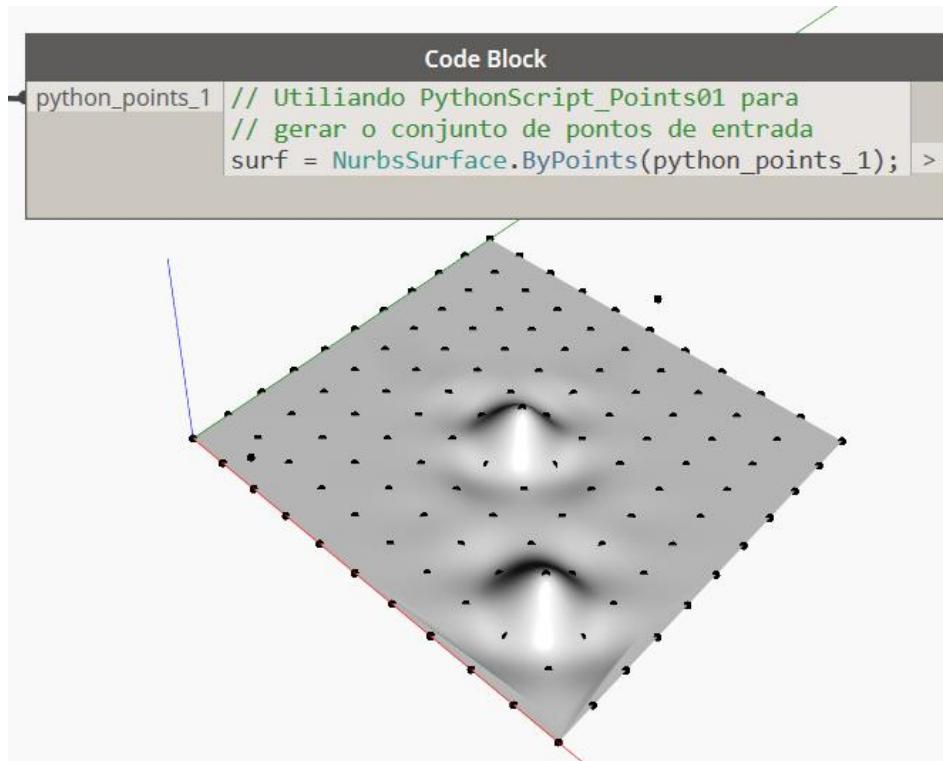
```

```
elif (i == 2 and j == 6):  
    z = 2  
    sub_points. (Point. (i, j, z))  
out_points. (sub_points)  
  
OUT = out_points
```

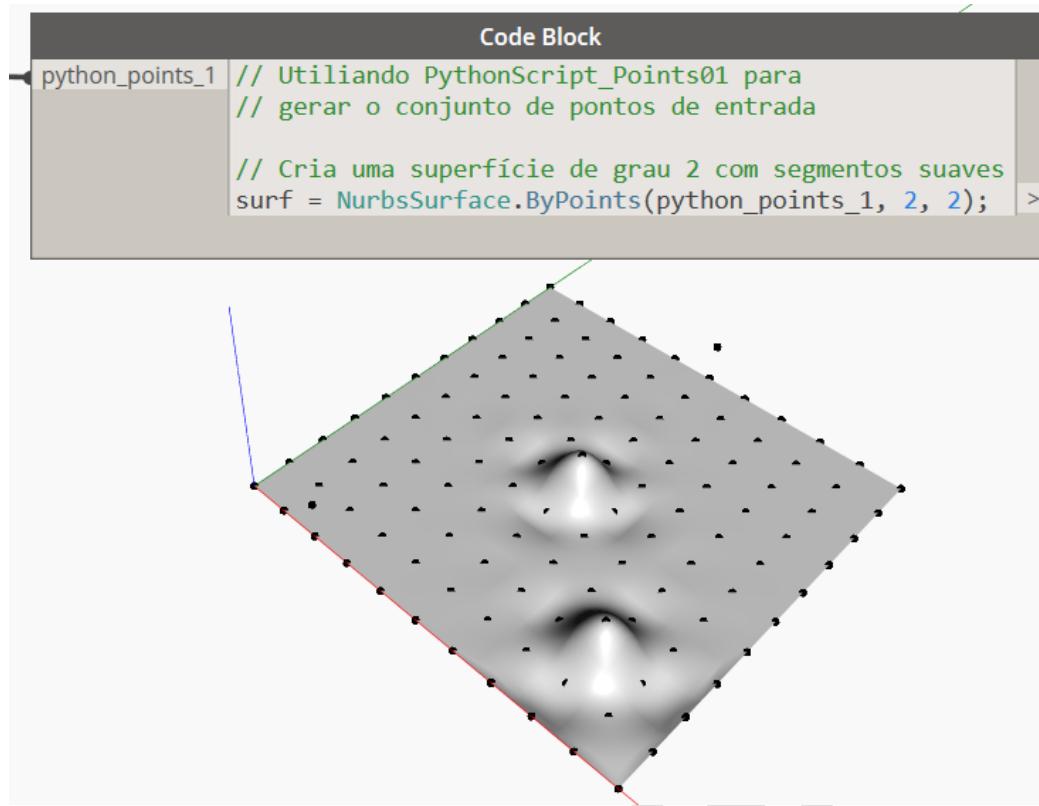
Superfícies: Interpretado, Pontos de Controle, Loft, Revolve

O analógico bidimensional de um NurbsCurve é o NurbsSurface e, como o NurbsCurve de forma livre, o NurbsSurfaces pode ser construído com dois métodos básicos: inserir um conjunto de pontos de base e fazer com que o Dynamo interprete entre eles, e especificar explicitamente os pontos de controle da superfície. Também como curvas de forma livre, as superfícies interpretadas são úteis quando um projetista sabe exatamente a forma que uma superfície precisa assumir ou se um projeto exige que a superfície passe por pontos de restrição. Por outro lado, as superfícies criadas por pontos de controle podem ser mais úteis para projetos exploratórios em vários níveis de suavização.

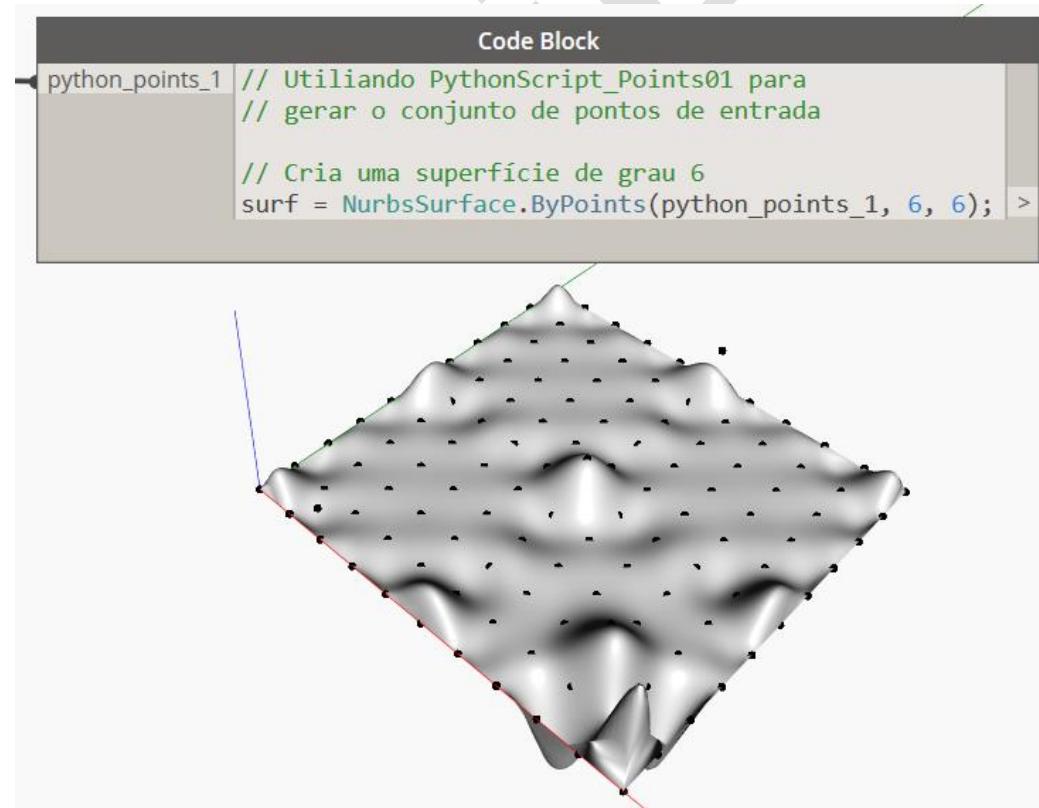
Para criar uma superfície interpretada, basta gerar uma coleção bidimensional de pontos que se aproxime da forma de uma superfície. A coleção deve ser retangular, ou seja, não serrilhada. O método NurbsSurface.ByPoints constrói uma superfície a partir desses pontos.



O NurbsSurfaces de forma livre também pode ser criado especificando pontos de controle subjacentes de uma superfície. Como o NurbsCurves, os pontos de controle podem ser vistos como representando uma malha quadrilateral com segmentos retos, que, dependendo do grau da superfície, são suavizados na forma final da superfície. Para criar um NurbsSurface por pontos de controle, inclua dois parâmetros adicionais para NurbsSurface.ByPoints, indicando os graus das curvas subjacentes nas duas direções da superfície.

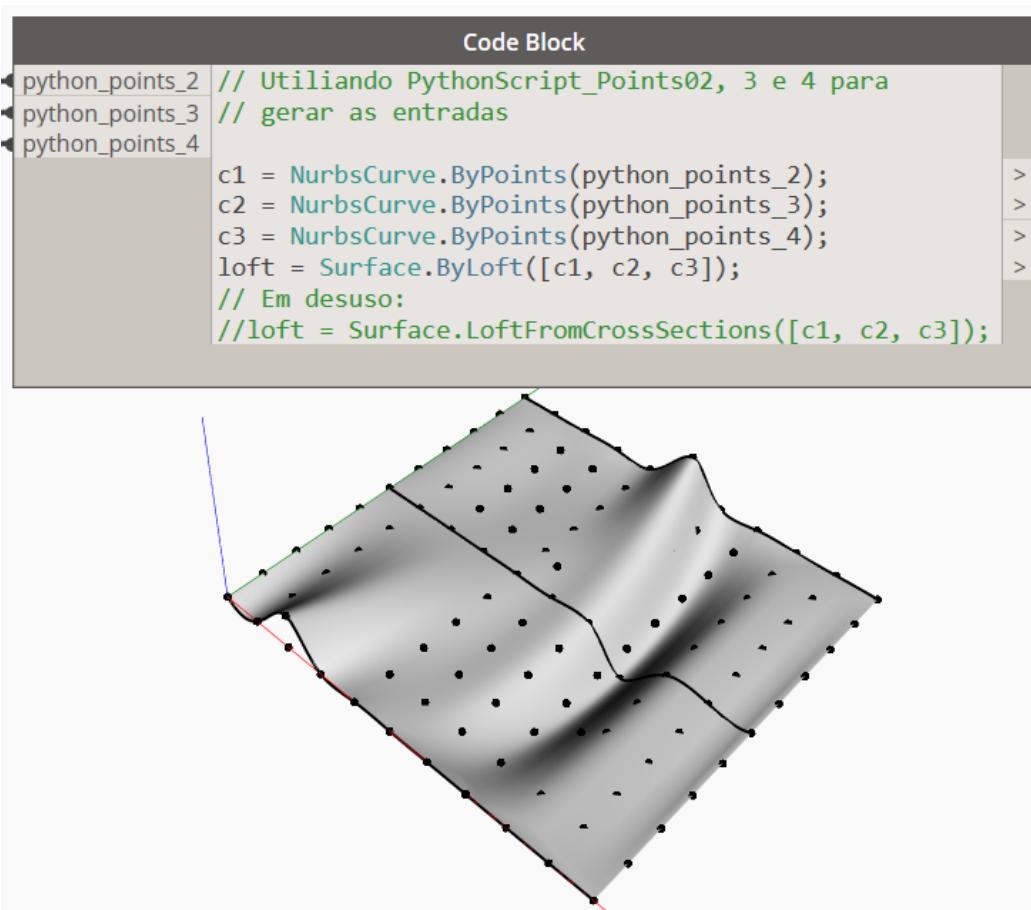


Podemos aumentar o grau do NurbsSurface para alterar a geometria da superfície resultante:

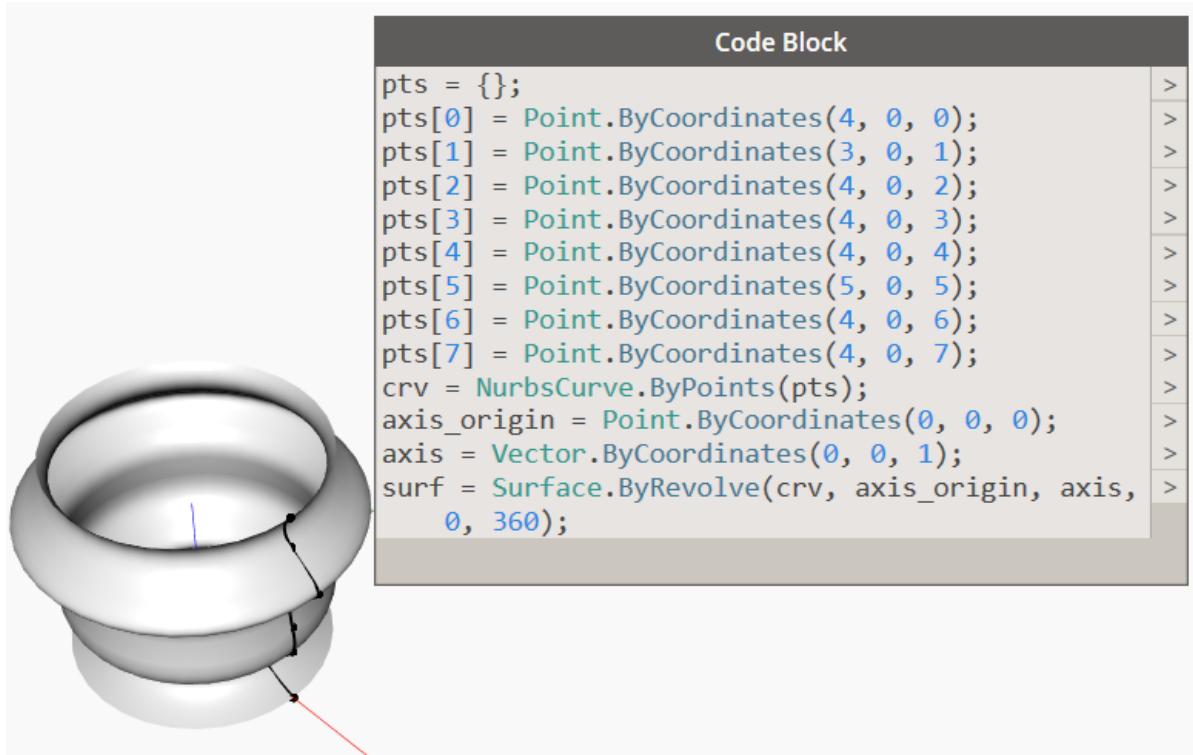


Assim como as superfícies podem ser criadas interpretando entre um conjunto de pontos de entrada, elas podem ser criadas interpretando entre um conjunto de curvas base. Isso se chama lofting.

Uma curva elevada é criada usando o construtor Surface.ByLoft, com uma coleção de curvas de entrada como o único parâmetro.



As superfícies de revolução são um tipo adicional de superfície criado pela varredura de uma curva base em torno de um eixo central. Se superfícies interpretadas são o análogo bidimensional das curvas interpretadas, as superfícies de revolução são o análogo bidimensional dos círculos e arcos. As superfícies de revolução são especificadas por uma curva base, representando a "aresta" da superfície; uma origem de eixo, o ponto base da superfície; uma direção do eixo, a direção central do "núcleo"; um ângulo de início da varredura; e um ângulo final de varredura. Eles são usados como entrada para o construtor Surface.Revolve.



Parametrização Geométrica

Em projetos computacionais, curvas e superfícies são frequentemente usadas como andaime subjacente para construir geometria subsequente. Para que essa geometria inicial seja usada como base para geometria posterior, o script deve ser capaz de extrair qualidades como posição e orientação em toda a área do objeto. Ambas as curvas e superfícies suportam essa extração, e isso é chamado de parametrização.

Todos os pontos em uma curva podem ser considerados como tendo um parâmetro exclusivo que varia de 0 a 1. Se criarmos um NurbsCurve com base em vários pontos de controle ou interpretados, o primeiro ponto terá o parâmetro 0 e o último point teria o parâmetro 1. É impossível saber antecipadamente qual é o parâmetro exato, qualquer ponto intermediário, que pode parecer uma limitação severa, embora seja mitigado por uma série de funções utilitárias. As superfícies têm uma parametrização semelhante às curvas, embora com dois parâmetros em vez de um, chamados u e v . Se criarmos uma superfície com os seguintes pontos:



p1 teria o parâmetro $u = 0$ $v = 0$, enquanto p9 teria os parâmetros $u = 1$ $v = 1$.

A parametrização não é particularmente útil ao determinar pontos usados para gerar curvas, seu principal uso é determinar os locais se pontos intermediários gerados pelos construtores NurbsCurve e NurbsSurface.

As curvas têm um método PointAtParameter, que utiliza um único argumento duplo entre 0 e 1 e retorna o objeto Point nesse parâmetro. Por exemplo, este script localiza os Pontos nos parâmetros 0, .1, .2, .3, .4, .5, .6, .7, .8, .9 e 1:

Code Block

```

pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(6, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(3, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
crv = NurbsCurve.ByPoints(pts);
pts_at_param = crv.PointAtParameter(0..1..#11);
// Desenha linhas para auxiliar na visualização
// dos pontos
lines = Line.ByStartPointEndPoint(pts_at_param, >
Point.ByCoordinates(4, 6, 0));

```

List

```

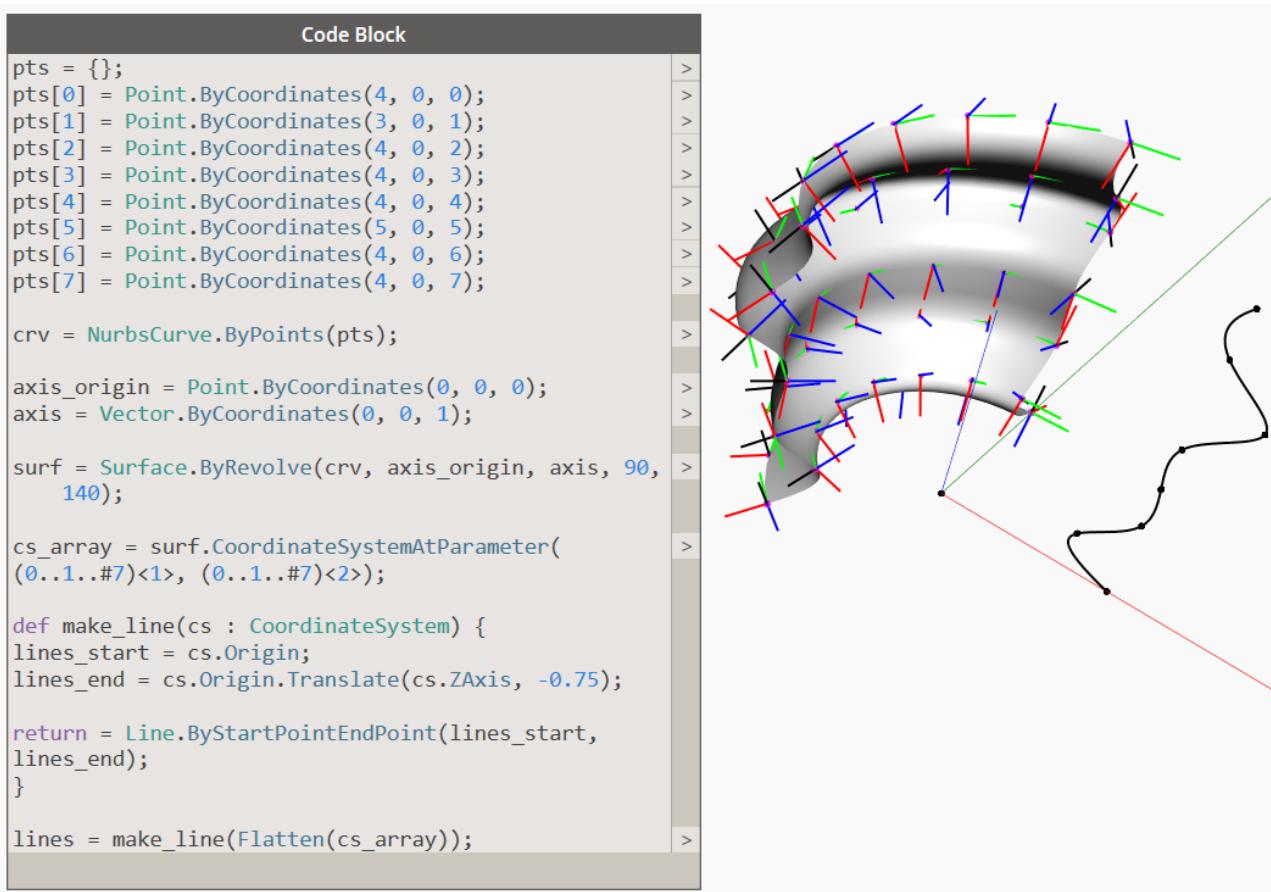
0 Line(StartPoint = Point(X = 4.000, Y = 0.000, Z = 0.000),
1 Line(StartPoint = Point(X = 5.928, Y = 0.000, Z = 0.582),
2 Line(StartPoint = Point(X = 6.203, Y = 0.000, Z = 0.902),
3 Line(StartPoint = Point(X = 5.508, Y = 0.000, Z = 1.147),
4 Line(StartPoint = Point(X = 4.530, Y = 0.000, Z = 1.503),
5 Line(StartPoint = Point(X = 3.950, Y = 0.000, Z = 2.156),
6 Line(StartPoint = Point(X = 4.020, Y = 0.000, Z = 3.116),
7 Line(StartPoint = Point(X = 3.984, Y = 0.000, Z = 4.033),
8 Line(StartPoint = Point(X = 3.305, Y = 0.000, Z = 4.718),
9 Line(StartPoint = Point(X = 2.898, Y = 0.000, Z = 5.307),
10 Line(StartPoint = Point(X = 4.000, Y = 0.000, Z = 6.000).

```

①L2 ②L1 {11}

Da mesma forma, as *Surfaces* têm um método *PointAtParameter* que aceita dois argumentos, o parâmetro *u* e *v* do *Point* gerado.

Embora a extração de pontos individuais em uma curva e superfície possa ser útil, os scripts geralmente exigem o conhecimento das características geométricas específicas de um parâmetro, como a direção da curva ou da superfície. Os métodos *CoordinateSystemAtParameterAlongCurve* e *CoordinateSystemAtParameter* encontram não apenas a posição, mas um *CoordinateSystem* orientado no parâmetro de uma *Curve* e *Surface*, respectivamente. Por exemplo, o script a seguir extrai o *CoordinateSystems* orientado ao longo de uma superfície revolvida e usa a orientação do *CoordinateSystems* para gerar linhas que se destacam normalmente na superfície:



Como mencionado anteriormente, a parametrização nem sempre é uniforme no comprimento de uma curva ou superfície, o que significa que o parâmetro 0,5 nem sempre corresponde ao ponto médio e 0,25 nem sempre corresponde ao ponto um quarto ao longo de uma curva ou superfície. Para contornar essa limitação, o Curves possui um conjunto adicional de comandos de parametrização que permitem encontrar um ponto em comprimentos específicos ao longo de uma Curve.

Interseção e Trim

Muitos dos exemplos até agora se concentraram na construção de geometria dimensional superior a partir de objetos dimensionais inferiores. Os métodos de interseção permitem que essa geometria dimensional mais alta gere objetos de dimensões mais baixas, enquanto os comandos *trim* e *select trim* permitem que o script modifique fortemente as formas geométricas após a criação.

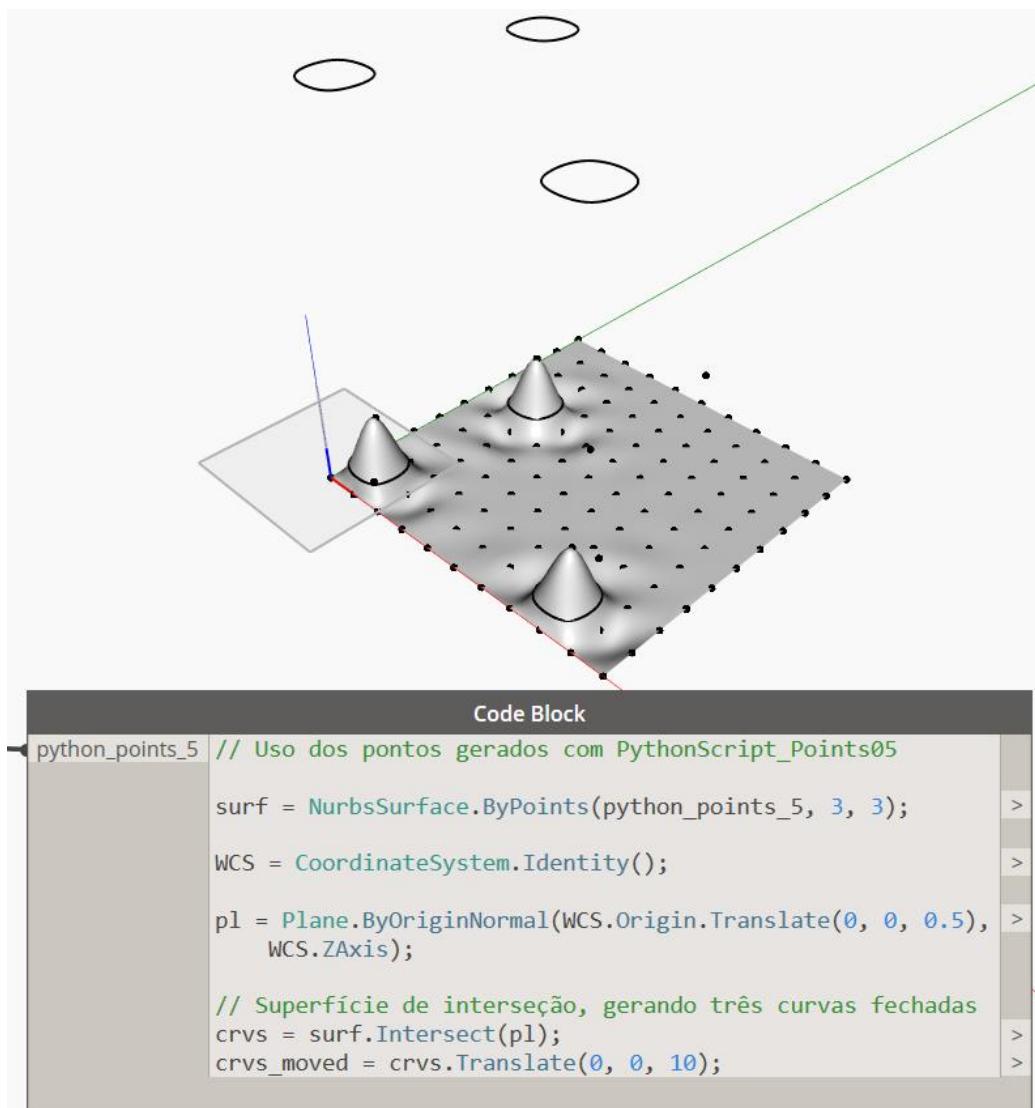
O método *Intersect* é definido em todas as partes da geometria no Dynamo, o que significa que, em teoria, qualquer parte da geometria pode ser cruzada com qualquer outra parte da geometria.

Naturalmente, algumas interseções não têm sentido, como interseções envolvendo pontos, pois o objeto resultante sempre será o próprio ponto de entrada. As outras combinações possíveis de interseções entre objetos estão descritas na tabela a seguir. A tabela a seguir descreve o resultado de várias operações de interseção:

Interseção:

Com:	Surface	Curve	Plane	Solid
Surface	Curve	Point	Point, Curve	Surface
Curve	Point	Point	Point	Curve
Plane	Curve	Point	Curve	Curve
Solid	Surface	Curve	Curve	Solid

O exemplo muito simples a seguir demonstra a interseção de um plano com um NurbsSurface. A interseção gera uma matriz NurbsCurve, que pode ser usada como qualquer outro NurbsCurve.

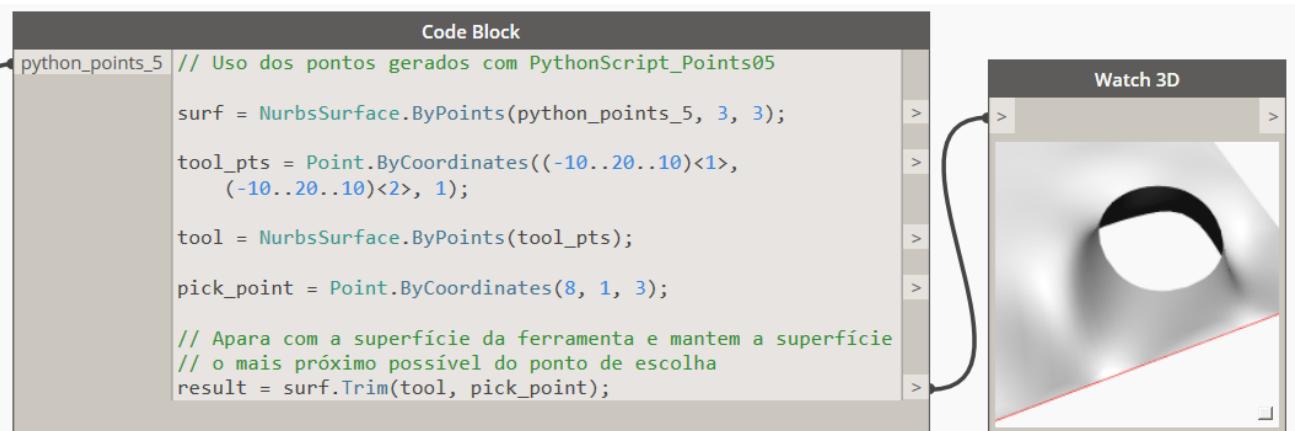


O método *Trim* é muito semelhante ao método *Intersect*, pois é definido para quase todas as partes da geometria. Ao contrário da interseção, há muito mais limitações no *Trim* do que no *Intersect*.

Trim:

Em:	Point	Curve	Plane	Surface	Solid
Curve	Sim	Não	Não	Não	Não
Polygon	NA	Não	Sim	Não	Não
Surface	NA	Sim	Sim	Sim	Sim
Solid	NA	NA	Sim	Sim	Sim

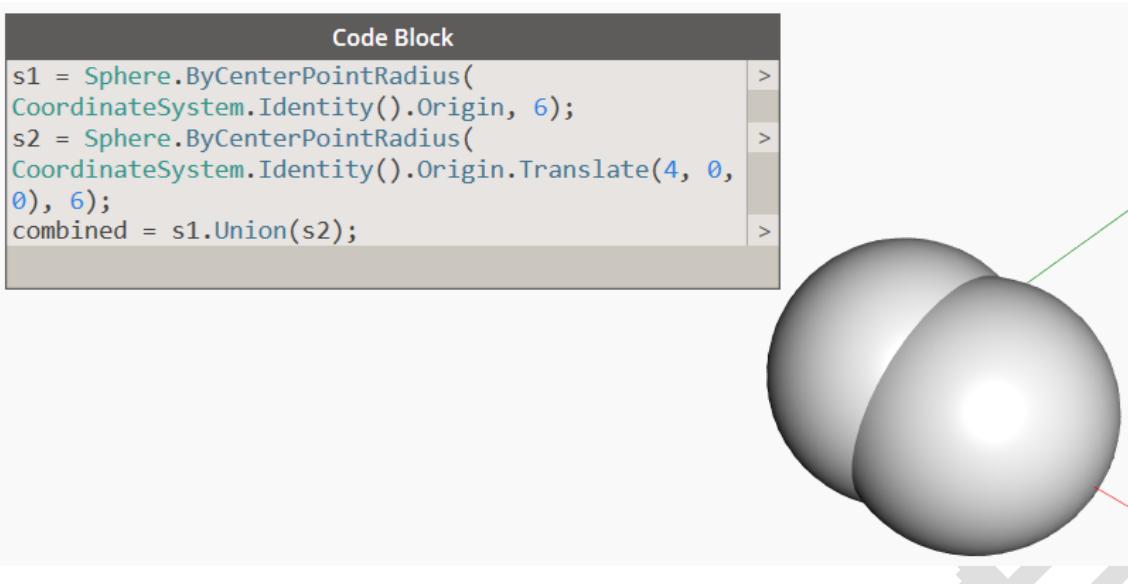
Algo a se observar sobre os métodos *Trim* é o requisito de um ponto "select", um ponto que determina qual geometria descartar e quais peças manter. O Dynamo encontra o lado mais próximo da geometria aparada até o ponto de seleção e esse lado se torna o lado a ser descartado.



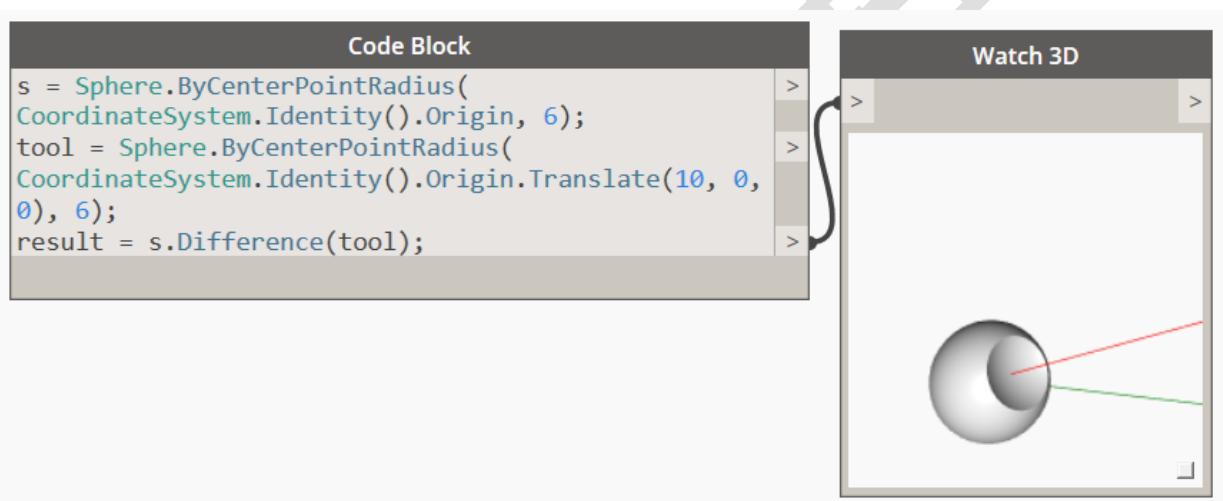
Booleanos Geométricos

Intersect, *Trim* e *SelectTrim* são usados principalmente em geometria de dimensões inferiores, como *Points*, *Curves* e *Surfaces*. A geometria sólida, por outro lado, possui um conjunto adicional de métodos para modificar a forma após sua construção, tanto subtraindo o material de maneira semelhante ao *Trim* e combinando elementos para formar um todo maior.

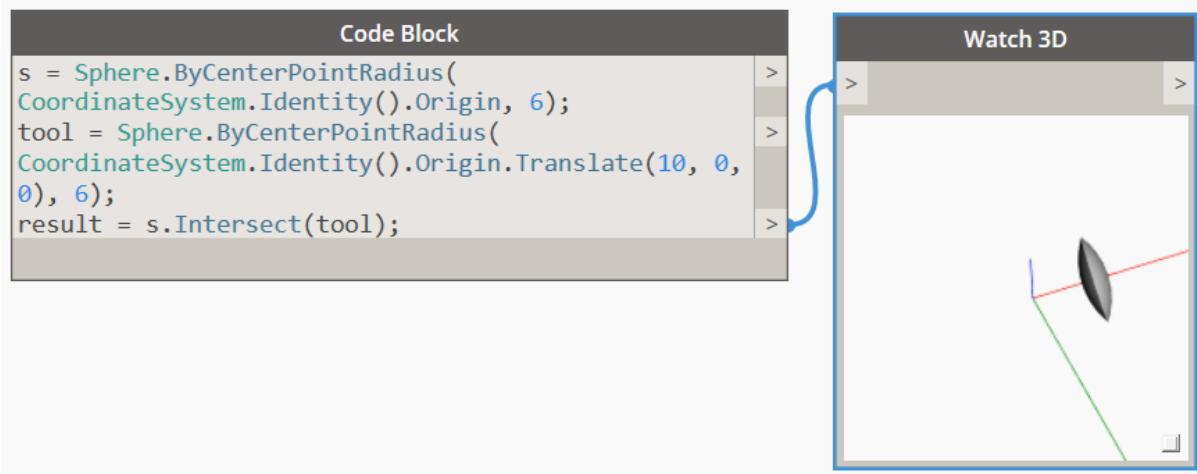
O método *Union* pega dois objetos sólidos e cria um único objeto sólido fora do espaço coberto por ambos os objetos. O espaço sobreposto entre objetos é combinado na forma final. Este exemplo combina uma esfera e um cubóide em uma única forma sólida de cubo de esfera:



O método *Difference*, como *Trim*, subtrai o conteúdo do sólido da ferramenta de entrada do sólido base. Neste exemplo, esculpimos um pequeno recuo de uma esfera:



O método *Intersect* retorna o sólido sobreposto entre duas entradas sólidas. No exemplo a seguir, a diferença foi alterada para *Intersect*, e o sólido resultante é o vazio que falta inicialmente entalhado:



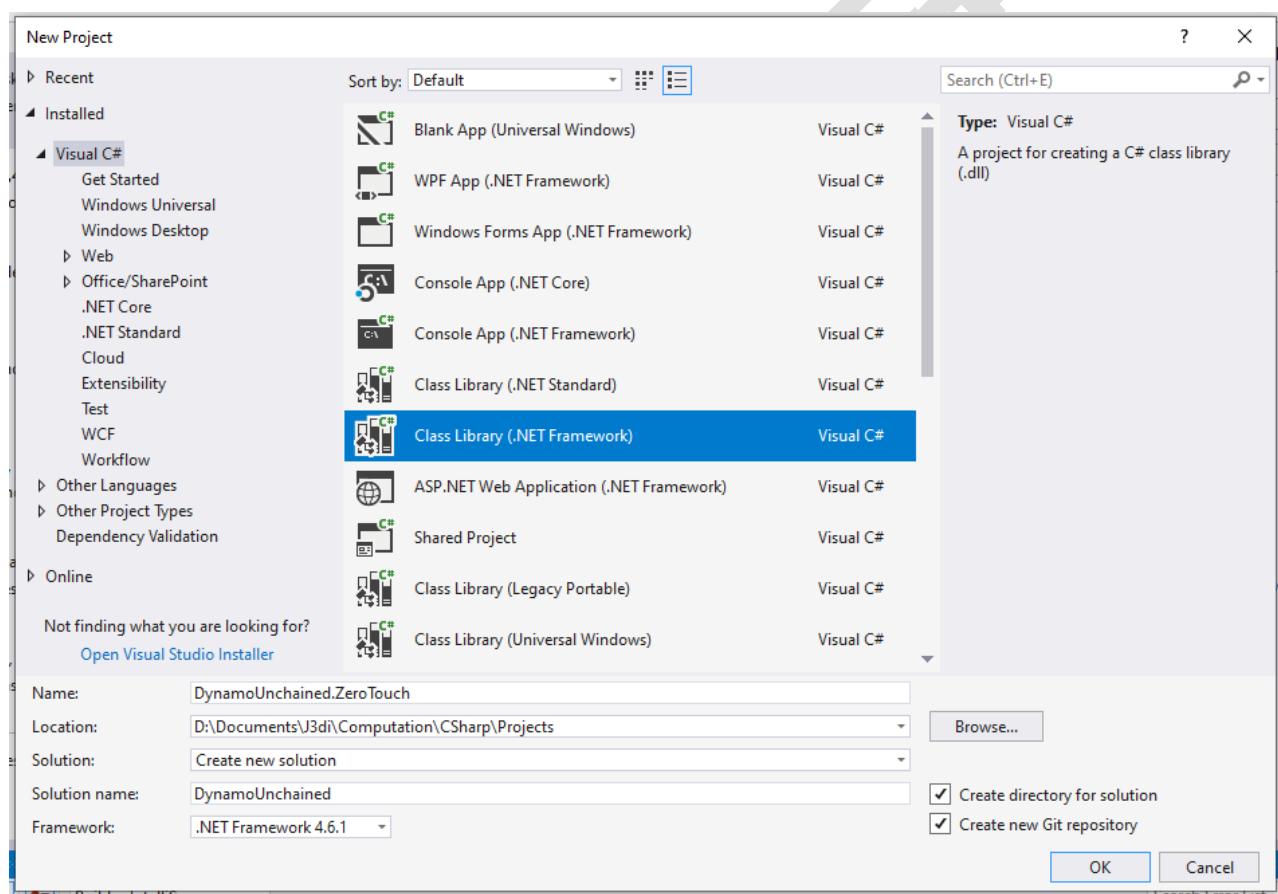
eTlipse

Criação de Módulo Dynamo no Visual Studio com C#

Adiante a melhor descrição e prática apresentadas possibilitarão melhor entendimento do Visual Studio utilizando C#. Por ora será apresentado o suficiente para o desenvolvimento de módulos do Dynamo personalizados com o uso do C# no Visual Studio.

É possível personalizar, criar, módulos do Dynamo no Visual Studio, utilizando a linguagem C#, neste caso usando a abordagem do Zero Touch Node. Embora possa trazer mais complexidade, o Visual Studio ajuda no desenvolvimento eficiente.

Criaremos um novo projeto, nome `DynamoUnchained.ZeroTouch` e solução `DynamoUnchained`, de uma biblioteca no Visual Studio.



Precisamos adicionar as referências adequadas. Iniciaremos gerenciando os pacotes NuGet. Botão direito do mouse na solução no Solution Explorer. Em seguida adicionaremos

DynamoVisualProgramming.ZeroTouchLibrary e DynamoVisualProgramming.DynamoServices. Atenção para a versão apropriada.

Solution Explorer

Search Solution Explorer (Ctrl+Shift+F)

Solution 'DynamoUnchained' (1 project)

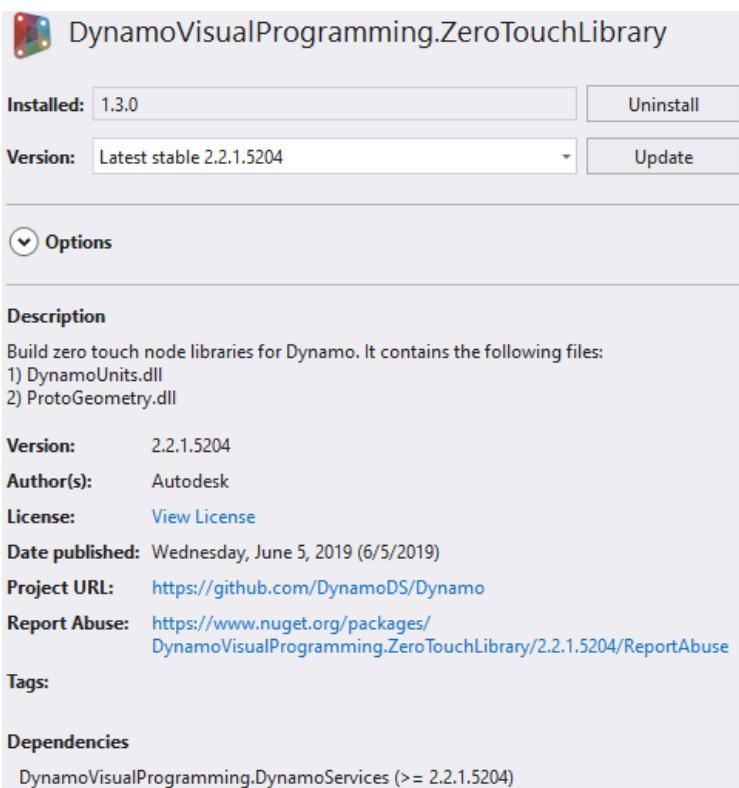
ZeroTouch

- Build
- Rebuild
- Clean
- Analyze
- Pack
- Manage NuGet Packages...**
- Scope to This
- New Solution Explorer View
- Show on Code Map
- Edit DynamoUnchained.ZeroTouch.csproj
- Add
- Cut Ctrl+X
- Remove Del
- Rename
- Unload Project
- Open Folder in File Explorer
- Properties Alt+Enter

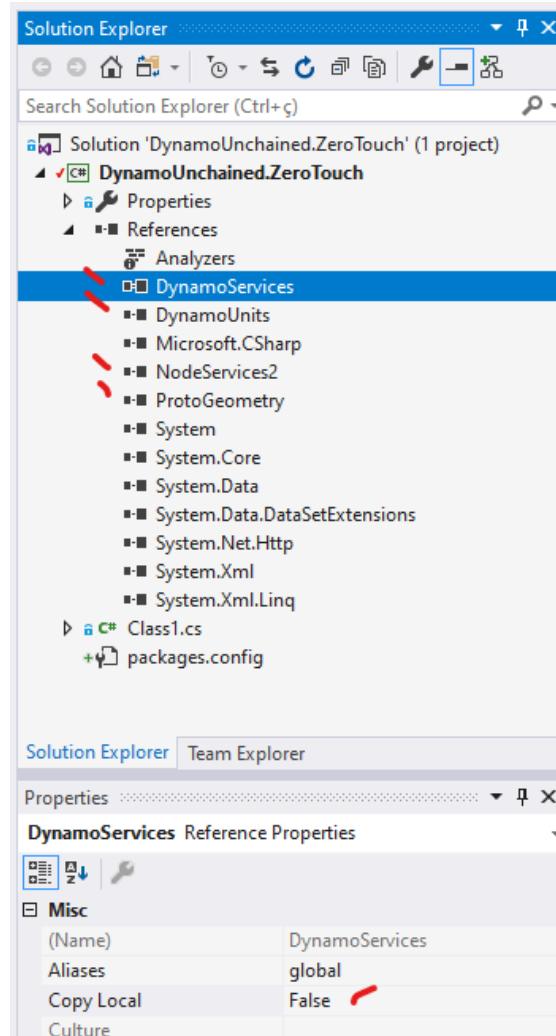
Browse Installed Updates

DynamoVisualProgramming Include prerelease

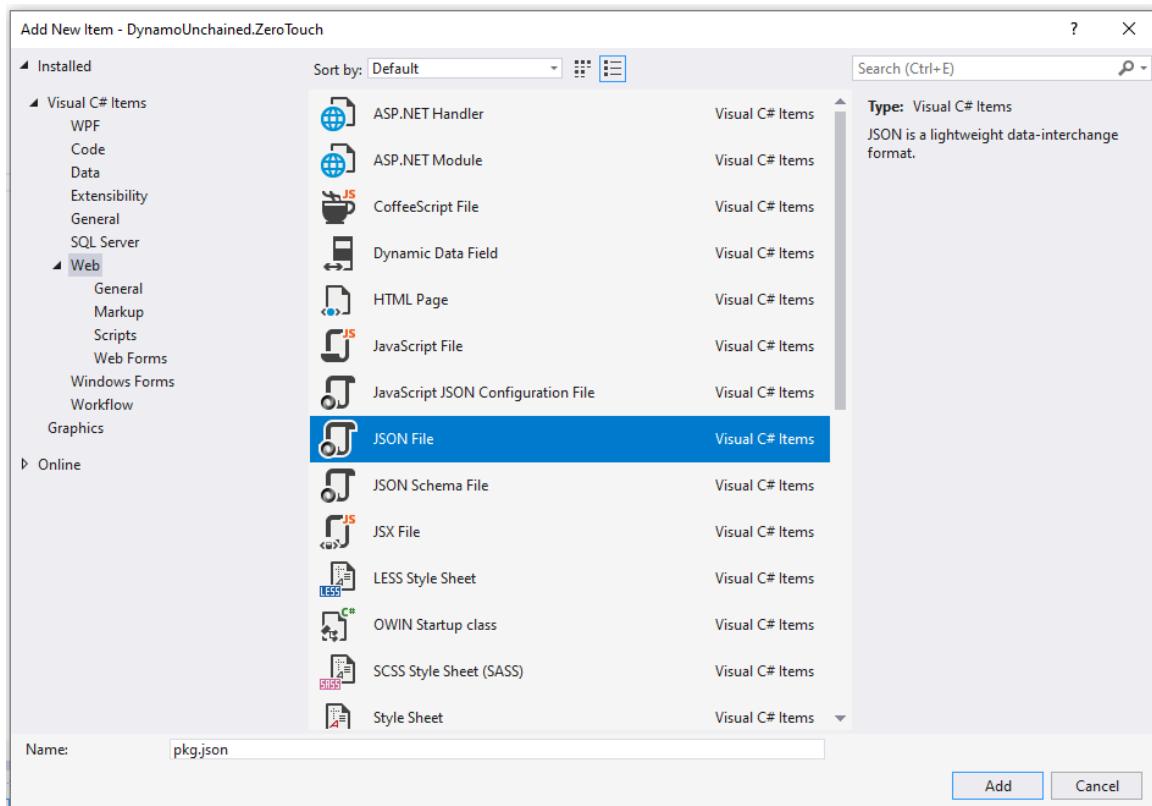
	DynamoVisualProgramming.ZeroTouchLibrary by Autodesk, 44.4K downloads	v2.2.1.5204
This package contains all that is required to get up and running building zero touch libraries for the Dynamo Visual Programming language.		
	DynamoVisualProgramming.WpfUILibrary by Autodesk, 23.4K downloads	v2.2.1.5204
This package contains all that is required to get up and running building nodes for the Dynamo Visual Programming language with custom UI in WPF.		
	DynamoVisualProgramming.DynamoCoreNodes by Autodesk, 15.7K downloads	v2.2.1.5204
Build core nodes for Dynamo. It contains the following files: 1) Analysis.dll		
	DynamoVisualProgramming.Core by Autodesk, 51.7K downloads	v2.2.1.5204
This package contains the core assemblies for Dynamo.		
	DynamoVisualProgramming.DynamoServices by Autodesk, 35.5K downloads	v2.2.1.5204
This package contains DynamoServices assembly that defines interfaces and attribute for Dynamo Zero Touch libraries.		



Quatro novas referências a bibliotecas são adicionadas, conforme visto no Solution Explorer. Para cada uma destas bibliotecas devemos configurar o parâmetro “Copy Local” para False, evitando assim que estas bibliotecas sejam copiadas para o destino durante a compilação.



Criaremos recursos para definição automática de pacote. Botão direito no projeto e adicionamos novo item, em seguida selecionamos JSON File, salvamos como pkg.json este arquivo. No arquivo copiamos o texto abaixo.



```
{
  "license": "",
  "file_hash": null,
  "name": "Dynamo Unchained - ZeroTouch",
  "version": "1.0.0",
  "description": "ZeroTouch sample node for the Dynamo Unchained workshop",
  "group": "",
  "keywords": null,
  "dependencies": [],
  "contents": "",
  "engine_version": "1.3.0.0",
  "engine": "dynamo",
  "engine_metadata": "",
  "site_url": "",
  "repository_url": "",
  "contains_binaries": true,
  "node_libraries": [
    "DynamoUnchained.ZeroTouch, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
  ]
}
```

Finalizamos a configuração do projeto com o procedimento para copiar automaticamente os arquivos para os pacotes do Dynamo, project>Properties>Build Events> Post-build event command line. Depois definimos que a compilação deve chamar o Revit, guia Debug, neste caso o caminho depende da versão do Revit instalada.

DynamoUnchained.ZeroTouch ➔ X | OlaDynamo.cs

Application Configuration: Active (Debug) Platform: Active (Any CPU)

Build

Build Events

Debug

Resources

Services

Settings

Reference Paths

Signing

Code Analysis

Start action

Start project

Start external program: C:\Program Files\Autodesk\Revit 2019\Revit.exe [Browse...](#)

Start browser with URL:

Start options

Command line arguments:

Working directory: [Browse...](#)

Use remote machine:

Debugger engines

Enable native code debugging

Enable SQL Server debugging

Build Events

Debug

Resources

Services

Settings

Reference Paths

Signing

Code Analysis

Pre-build event command line:

[Edit Pre-build...](#)

Post-build event command line:

```
getDir)*.*" "$(AppData)\Dynamo\Dynamo Revit\1.3\packages\$(ProjectName)\bin\"  
| jectDir)pkg.json" "$(AppData)\Dynamo\Dynamo Revit\1.3\packages\$(ProjectName)"
```

```
xcopy /Y "$(TargetDir)*.*" "$(AppData)\Dynamo\Dynamo  
Revit\1.3\packages\$(ProjectName)\bin\"
```

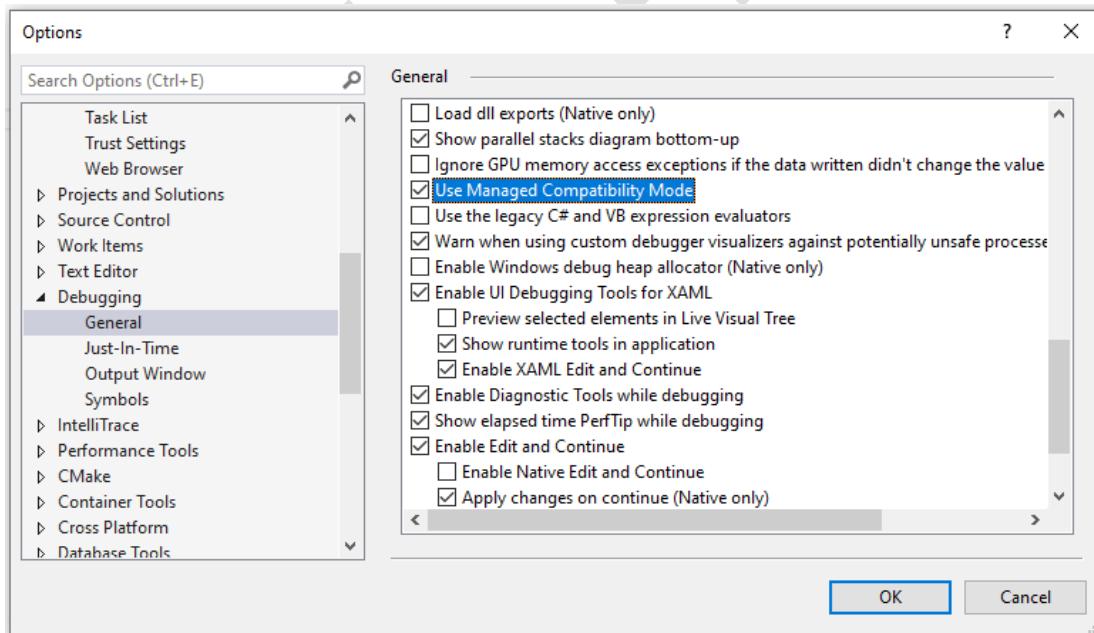
```
xcopy /Y "$(ProjectDir)pkg.json" "$(AppData)\Dynamo\Dynamo
Revit\1.3\packages\$(ProjectName)"
```

Exemplo - Olá Dynamo

Usaremos o tradicional programa inicial de “Olá” para testar nossa habilidade de gerar um pacote Dynamo. Deveremos renomear Class1.cs para OlaDynamo.cs. O código ficará como segue.

```
namespace DynamoUnchained.ZeroTouch
{
    public static class OlaDynamo{
        public static string DigaOla(string Nome) {
            return "Olá " + Nome + "!";
        }
    }
}
```

Precisamos de mais uma configuração para o recurso de Debug, Tools > Options... > Debugging > General > Check “Use Managed Compatibility Mode”.



Para mantermos o pacote Dynamo corretamente aninhado, adicionaremos um arquivo, DynamoUnchained.ZeroTouch_DynamoCustomization.xml, com o qual faremos a definição correta. O

arquivo ficará conforme abaixo, e devemos configurar o parâmetro Copy to Output Directory para Copy Always. Com esta última configuração poderemos compilar e testar o pacote. Podemos notar que o Dynamo considera métodos públicos estáticos como nós, e leva em conta os dados de entrada e saída do método igualmente nos nós.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>DynamoUnchained.ZeroTouch</name>
  </assembly>
  <namespaces>
    <namespace name="DynamoUnchained.ZeroTouch">
      <category>eTlipse_v_20190806</category>
    </namespace>
  </namespaces>
</doc>
```



Exemplo - Média

Testando um caso em que tenhamos mais de uma entrada, podemos fazer uso de uma função de média de duas entradas.

```
public static double MediaNumerica(double Numero1, double Numero2) {
  return ((Numero1 + Numero2) / 2);
}
```



Dica:

```
/*
```

Para aceitar qualquer tipo é possível utilizar “object”. Caso a estrutura seja desconhecida, devemos adicionar a diretiva “using Autodesk.DesignScript.Runtime;” e usar o atributo [ArbitraryDimensionArrayImport].

```
public static IList AddItemToEnd([ArbitraryDimensionArrayImport] object item,
IList list) {
    return new ArrayList(list) //Clone original list
    {
        item //Add item to the end of cloned list.
    };
}
```

```
*/
```

Exemplo - Saída de Vários Valores (Separa Par/ímpar)

É possível receber uma lista e dividir seus valores em grupos distintos para saída de mais de um valor ou conjunto de valores. Este exemplo recebe uma lista de números e os separa em duas listas, uma de ímpares e a outra de pares.

```
using Autodesk.DesignScript.Runtime;

[MultiReturn(new[] { "pares", "ímpares" })]
public static Dictionary<string, object> SeparaImparPar(List<int> list) {
    var pares = new List<int>();
    var ímpares = new List<int>();

    // Checa se o inteiro é par ou ímpar
```

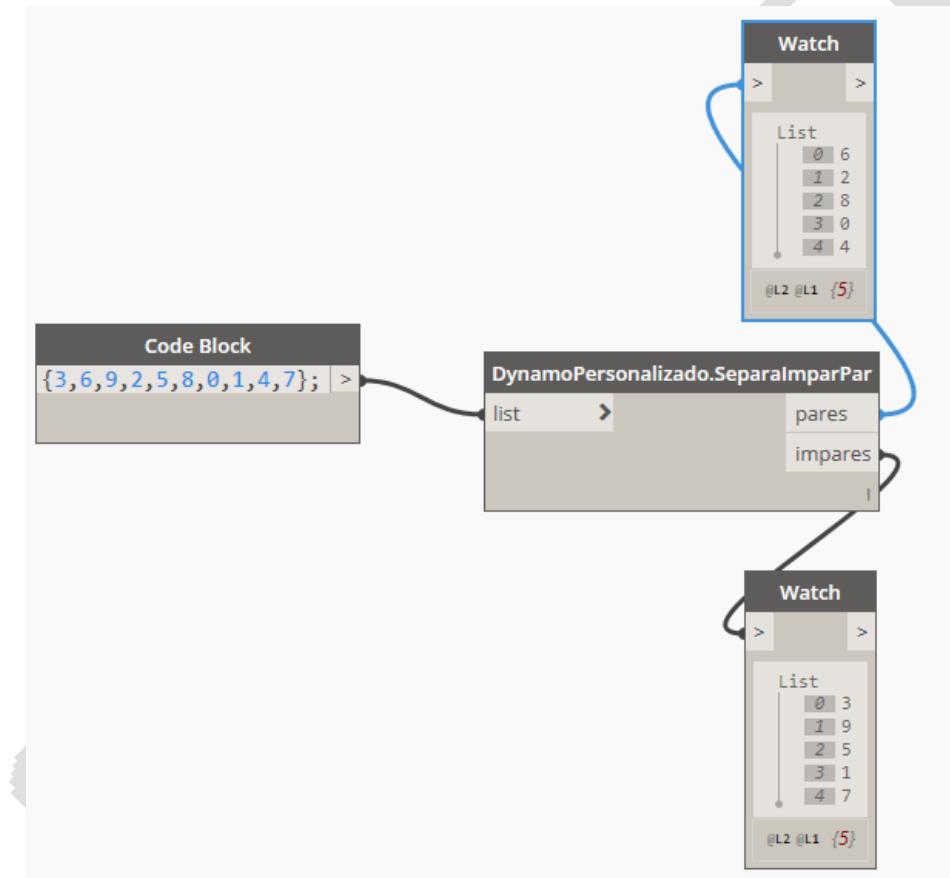
```

foreach (var i in list) {
    if (i % 2 == 0) {
        pares.Add(i);
    } else {
        impares.Add(i);
    }
}

// Cria um dicionário e o retorna
var d = new Dictionary<string, object>();
d.Add("pares", pares);
d.Add("impares", impares);
return d;
}
}

```

// O código acima pode ser simplificado em uma linha
//return new Dictionary<string, object> { { "par", par }, { "impar", impar } };
}



Exemplo - Geometria

Podemos fazer uso das implementações de geometria do Dynamo. Precisamos adicionar a diretiva “using Autodesk.DesignScript.Geometry”.

```
[MultiReturn(new[] { "X", "Y", "Z" })]
    public static Dictionary<string, object> PointCoordinates(Point point) {
        return new Dictionary<string, object> { { "X", point.X }, { "Y", point.Y },
    }, { "Z", point.Z } };
    }

    public static Line ByCoordinates(double X1, double Y1, double Z1, double X2,
double Y2, double Z2) {
        var p1 = Point.ByCoordinates(X1, Y1, Z1);
        var p2 = Point.ByCoordinates(X2, Y2, Z2);

        return Line.ByStartPointEndPoint(p1, p2);
    }
```

Atenção:

```
/*
```

Os elementos que não foram usados ou que não retornamos devem ser descartados corretamente. Podemos usar “dispose” ou encapsular o código em um bloco “using”.

```
public static Line ByCoordinatesReturnInteger(double X1, double Y1, double Z1,
double X2, double Y2, double Z2) {
    var p1 = Point.ByCoordinates(X1, Y1, Z1);
    var p2 = Point.ByCoordinates(X2, Y2, Z2);
    var l = Line.ByStartPointEndPoint(p1, p2);
    p1.Dispose();
    p2.Dispose();
    return l;
}

public static Line ByCoordinatesReturnLine(double X1, double Y1, double Z1,
double X2, double Y2, double Z2) {
    using (var p1 = Point.ByCoordinates(X1, Y1, Z1)) {
        using (var p2 = Point.ByCoordinates(X2, Y2, Z2)) {
            return Line.ByStartPointEndPoint(p1, p2);
        }
    }
}

/*
```

Até o momento nosso código completo, utilizando as funções descritas até este ponto, ficará:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.DesignScript.Runtime;
using Autodesk.DesignScript.Geometry;

namespace DynamoUnchained.ZeroTouch
{
    public static class DynamoPersonalizado{

        public static string DigaOla(string Nome) {
            return "Olá " + Nome + "!";
        }

        public static double MediaNumerica(double Numero1, double Numero2) {
            return ((Numero1 + Numero2) / 2);
        }

        [MultiReturn(new[] { "pares", "impares" })]
        public static Dictionary<string, object> SeparaImparPar(List<int> list) {
            var pares = new List<int>();
            var impares = new List<int>();

            // Checa se o inteiro é par ou ímpar
            foreach (var i in list) {
                if (i % 2 == 0) {
                    pares.Add(i);
                } else {
                    impares.Add(i);
                }
            }

            // Cria um dicionário e o retorna
            var d = new Dictionary<string, object>();
            d.Add("pares", pares);
            d.Add("impares", impares);
            return d;
        }

        // O código acima pode ser simplificado em uma linha
        //return new Dictionary<string, object> { { "par", par }, { "ímpar", ímpar } };

        [MultiReturn(new[] { "X", "Y", "Z" })]
        public static Dictionary<string, object> PointCoordinates(Point point) {
            return new Dictionary<string, object> { { "X", point.X }, { "Y", point.Y },
            { "Z", point.Z } };
        }

        public static Line ByCoordinates(double X1, double Y1, double Z1, double X2,
double Y2, double Z2) {
            var p1 = Point.ByCoordinates(X1, Y1, Z1);
            var p2 = Point.ByCoordinates(X2, Y2, Z2);
        }
    }
}

```

```
        return Line.ByStartPointEndPoint(p1, p2);
    }

    public static Line ByCoordinatesReturnInteger(double X1, double Y1, double Z1,
double X2, double Y2, double Z2) {
        var p1 = Point.ByCoordinates(X1, Y1, Z1);
        var p2 = Point.ByCoordinates(X2, Y2, Z2);
        var l = Line.ByStartPointEndPoint(p1, p2);
        p1.Dispose();
        p2.Dispose();
        return l;
    }

    public static Line ByCoordinatesReturnLine(double X1, double Y1, double Z1,
double X2, double Y2, double Z2) {
        using (var p1 = Point.ByCoordinates(X1, Y1, Z1)) {
            using (var p2 = Point.ByCoordinates(X2, Y2, Z2)) {
                return Line.ByStartPointEndPoint(p1, p2);
            }
        }
    }
}
```

Exemplo - Elementos do Revit

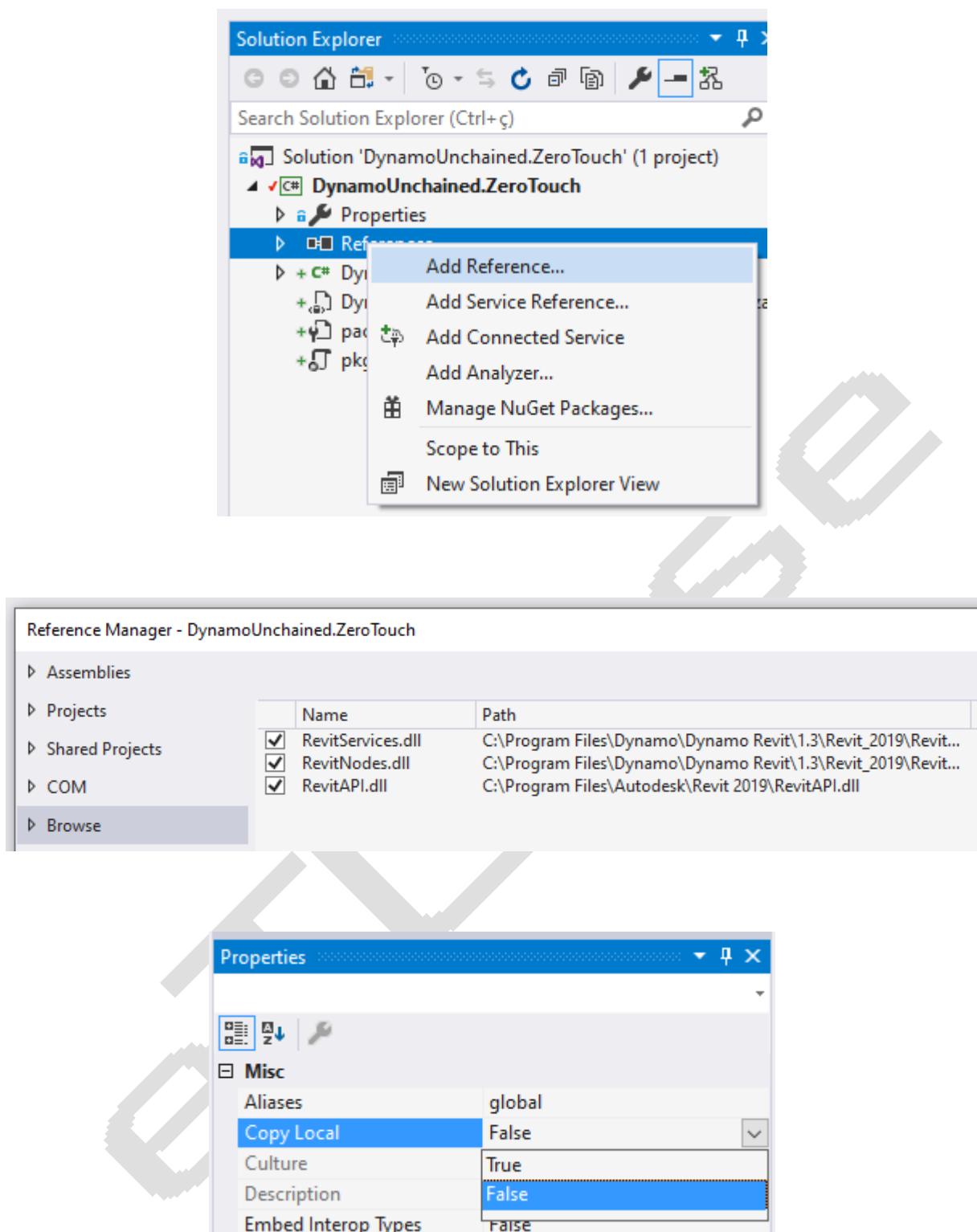
Existe diferença entre as classes do Revit e as do Dynamo, sendo necessário usar as do Revit para instanciar elementos nele. Neste exemplo é possível observar esse uso.

Precisamos adicionar mais 3 referências em nosso projeto, a primeira dll do Revit, e as duas seguintes dlls do Dynamo. Importante configurar a propriedade Copy Local para False, para estas dlls adicionadas. As bibliotecas devem ser utilizadas conforme a versão do Revit instalada. As bibliotecas são:

C:\Program Files\Autodesk\Revit 2019\RevitAPI.dll

C:\Program Files\Dynamo\Dynamo Revit\1.3\Revit_2019\RevitNodes.dll

C:\Program Files\Dynamo\Dynamo Revit\1.3\Revit_2019\RevitServices.dll



Devemos adicionar em nosso projeto, na classe base que estamos trabalhando, as diretivas para usar as classes adequadas:

```
using Autodesk.Revit.DB; using Revit.Elements; using RevitServices.Persistence;
```

Importante pontuar que Revit e Dynamo possuem elementos com o mesmo nome, de tal forma que é preciso informar a qual classe estamos nos referindo, isto pode ser feito utilizando o caminho completo da classe. Este é o caso de wall, que deve ser referenciado da seguinte forma:

```
public static Autodesk.Revit.DB.Wall wall1; // Revit wall
public static Revit.Elements.Wall wall2; // Dynamo wall
```

Para esta conversão entre o Revit e o Dynamo, podemos utilizar as referências a seguir.

Do Revit para Dynamo

```
//Elements
Element.ToDSType(bool); //true if it's an element generated by Revit
//Geometry
XYZToPoint() > Point
XYZToVector() > Vector
Point.ToProtoType() > Point
List<XYZ>.ToPoints() > List<Point>
UV.ToProtoType() > UV
Curve.ToProtoType() > Curve
CurveArray.ToProtoType() > PolyCurve
PolyLine.ToProtoType() > PolyCurve
Plane.ToPlane() > Plane
Solid.ToProtoType() > Solid
Mesh.ToProtoType() > Mesh
IEnumerable<Mesh>.ToProtoType() > Mesh[]
```

```
Face.ToProtoType() > IEnumerable<Surface>  
Transform.ToCoordinateSystem() > CoordinateSystem  
BoundingBoxXYZ.ToProtoType() > BoundingBox
```

Do Revit para Dynamo

```
//Elements  
  
Element.InternalElement  
  
//Geometry  
  
Point.ToRevitType() > XYZ  
Vector.ToRevitType() > XYZ  
Plane.ToPlane() > Plane  
List<Point>.ToXyzs() > List<XYZ>  
Curve.ToRevitType() > Curve  
PolyCurve.ToRevitType() > CurveLoop  
Surface.ToRevitType() > IList<GeometryObject>  
Solid.ToRevitType() > IList<GeometryObject>  
Mesh.ToRevitType() > IList<GeometryObject>  
CoordinateSystem.Transform() > Transform  
CoordinateSystem.ToRevitBoundingBox() > BoundingBoxXYZ  
BoundingBox.ToRevitType() > BoundingBoxXYZ
```

Finalmente podemos escrever nosso nó que recebe parede na entrada e retorna sua linha base.

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using Autodesk.DesignScript.Runtime;  
using Autodesk.DesignScript.Geometry;
```

```

using Autodesk.Revit.DB;
using Revit.Elements;
using RevitServices.Persistence;
using Revit.GeometryConversion;

namespace DynamoUnchained.ZeroTouch
{
    public static class DynamoPersonalizado{

        public static Autodesk.DesignScript.Geometry.Curve
GetWallBaseline(Revit.Elements.Wall wall) {
            //get Revit Wall
            var revitWall = wall.InternalElement;
            //revit API
            var locationCurve = revitWall.Location as LocationCurve;
            //convert to Dynamo and return it
            return locationCurve.Curve.ToProtoType();
        }

    }
}

```

Exemplo - Parede desenha texto

Este exemplo recebe um texto, texto este que é convertido em linhas, e em seguida utiliza as linhas geradas para base de uma parede.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.DesignScript.Runtime;
using Autodesk.DesignScript.Geometry;
using Autodesk.Revit.DB;
using Revit.Elements;
using RevitServices.Persistence;
using Revit.GeometryConversion;
using System.Drawing;
using System.Drawing.Drawing2D;

namespace DynamoUnchained.ZeroTouch
{
    public static class DynamoPersonalizado{

        public static Autodesk.DesignScript.Geometry.Curve
GetWallBaseline(Revit.Elements.Wall wall) {
            //get Revit Wall
            var revitWall = wall.InternalElement;
            //revit API
            var locationCurve = revitWall.Location as LocationCurve;
            //convert to Dynamo and return it
            return locationCurve.Curve.ToProtoType();
        }

    }
}

```

```

    /// <summary>
    /// Converte uma em uma lista de segmentos
    /// </summary>
    /// <param name="text">String a ser convertida</param>
    /// <param name="size">Tamanho do texto</param>
    /// <returns></returns>
    [IsVisibleInDynamoLibrary(false)]
    public static IEnumerable<Line> TextToLines(string text, int size) {
        List<Line> lines = new List<Line>();

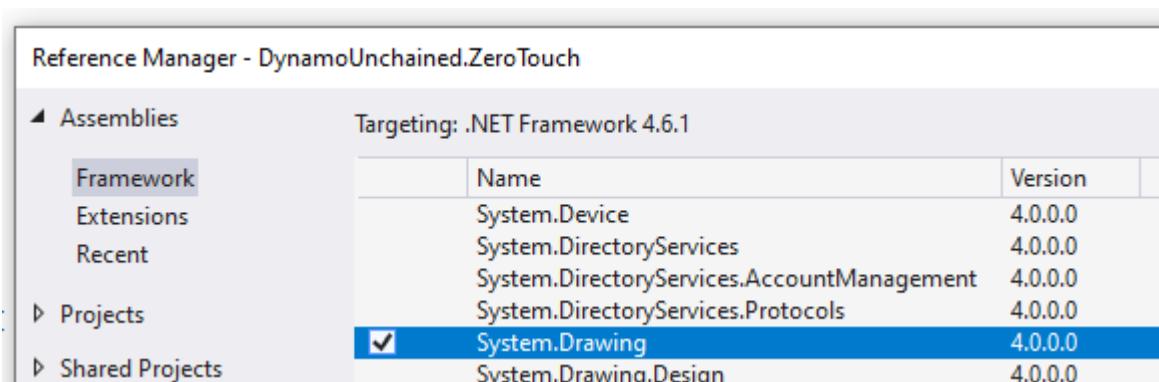
        //using System.Drawing para conversão para pontos da fonte
        using (Font font = new System.Drawing.Font("Arial", size,
        FontStyle.Regular))
            using (GraphicsPath gp = new GraphicsPath())
                using (StringFormat sf = new StringFormat()) {
                    sf.Alignment = StringAlignment.Center;
                    sf.LineAlignment = StringAlignment.Center;

                    gp.AddString(text, font.FontFamily, (int)font.Style, font.Size, new
                    PointF(0, 0), sf);
                    // Converte os pontos da fonte para pontos do Dynamo
                    var points = gp.PathPoints.Select(p =>
                    Autodesk.DesignScript.Geometry.Point.ByCoordinates(p.X, -p.Y, 0)).ToList();
                    var types = gp.PathTypes;

                    Autodesk.DesignScript.Geometry.Point start = null;
                    // Cria linhas
                    for (var i = 0; i < types.Count(); i++) {
                        //Types:
                        //0 início de uma forma
                        //1 ponto em linha
                        //3 ponto em curva
                        //129 fim parcial de linha
                        //131 fim parcial de curva
                        //161 fim de linha
                        //163 fim de curva
                        if (types[i] == 0) {
                            start = points[i];
                        }
                        // Algumas letras precisam ser fechadas e outras não
                        if (types[i] > 100) {
                            if (!points[i].IsAlmostEqualTo(start)) {
                                lines.Add(Line.ByStartPointEndPoint(points[i], start));
                            }
                        } else {
                            lines.Add(Line.ByStartPointEndPoint(points[i], points[i +
                            1]));
                        }
                    }
                    // Descartar os pontos
                    foreach (var point in points) {
                        point.Dispose();
                    }
                    return lines;
                }
            }
        }
    }
}

```

Usamos a diretiva `[IsVisibleInDynamoLibrary(false)]` para evitar a exibição no Dynamo.
 Precisamos adicionar a referência a System.Drawing em References>Add Reference...>Assemblies>Framework.



O código completo será conforme segue, e em seguida podemos visualizar uma possível saída usando o nó no Dynamo.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Drawing;
using System.Drawing.Drawing2D;
using Autodesk.Revit.DB; //Revit API
using Revit.Elements; //Dynamo Revit Elements
using RevitServices.Persistence;
using RevitServices.Transactions;
using Revit.GeometryConversion;
using Autodesk.DesignScript.Runtime;

namespace DynamoUnchained.ZeroTouch {
    public static class DynamoPersonalizado {
        public static Autodesk.DesignScript.Geometry.Curve
GetWallBaseline(Revit.Elements.Wall wall) {
            // Pegar Revit Wall
            var revitWall = wall.InternalElement;
            // Revit API
            var locationCurve = revitWall.Location as LocationCurve;
            // Converte para Dynamo e o retorna
            return locationCurve.Curve.ToProtoType();
        }
    }
}
```

```

    /// <summary>
    /// Text to Wall gera paredes Revit baseado na entrada de um texto para
orientar as curvas
    /// </summary>
    /// <param name="text">Texto para converter em linhas base Wall</param>
    /// <param name="height"> Altura da Parede</param>
    /// <param name="level"> Nivel da Parede</param>
    /// <param name="wallType"> Tipo de Parede</param>
    /// <param name="size"> Tamanho da Fonte</param>
    /// <returns></returns>
public static IEnumerable<Revit.Elements.Wall> TextToWall(string text, double
height, Revit.Elements.Level level, Revit.Elements.WallType wallType, int size = 25) {
    // Primeiro checar as entradas
    if (level == null) {
        throw new ArgumentNullException("level");
    }

    if (wallType == null) {
        throw new ArgumentNullException("wallType");
    }

    var walls = new List<Revit.Elements.Wall>();
    var lines = DynamoPersonalizado.TextToLines(text, size);

    // Criação e modificação de elementos devem estar dentro de uma transação
    TransactionManager.Instance.EnsureInTransaction(Document);

    foreach (var curve in lines) {
        if (curve == null) {
            throw new ArgumentNullException("curve");
        }

        try {
            var wall = Autodesk.Revit.DB.Wall.Create(Document,
curve.ToRevitType(), wallType.InternalElement.Id, level.InternalElement.Id, height,
0.0, false, false);
            walls.Add(wall.ToDSType(false) as Revit.Elements.Wall);
        } catch (Exception ex) {
            throw new ArgumentException(ex.Message);
        }
    }
}

TransactionManager.Instance.TransactionTaskDone();

return walls;
}

internal static Autodesk.Revit.DB.Document Document {
    get { return DocumentManager.Instance.CurrentDBDocument; }
}

    /// <summary>
    /// Converte uma string em uma lista de segmentos
    /// </summary>
    /// <param name="text"> String para converter</param>
    /// <param name="size"> Tamanho do texto</param>
    /// <returns></returns>
[IsVisibleInDynamoLibrary(false)]
public static IEnumerable<Autodesk.DesignScript.Geometry.Line>
TextToLines(string text, int size) {

```

```
    List<Autodesk.DesignScript.Geometry.Line> lines = new
    List<Autodesk.DesignScript.Geometry.Line>();

        //using System.Drawing para conversão em pontos da fonte
        using (Font font = new System.Drawing.Font("Arial", size,
FontStyle.Regular))
            using (GraphicsPath gp = new GraphicsPath())
                using (StringFormat sf = new StringFormat()) {
                    sf.Alignment = StringAlignment.Center;
                    sf.LineAlignment = StringAlignment.Center;

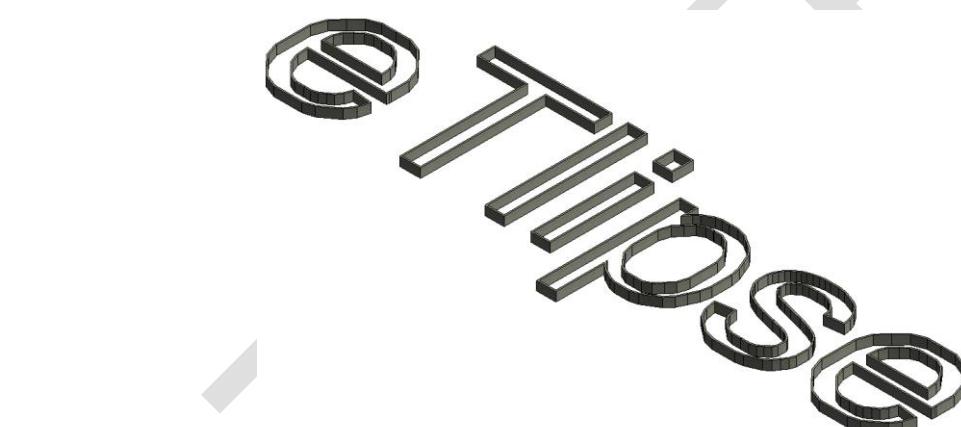
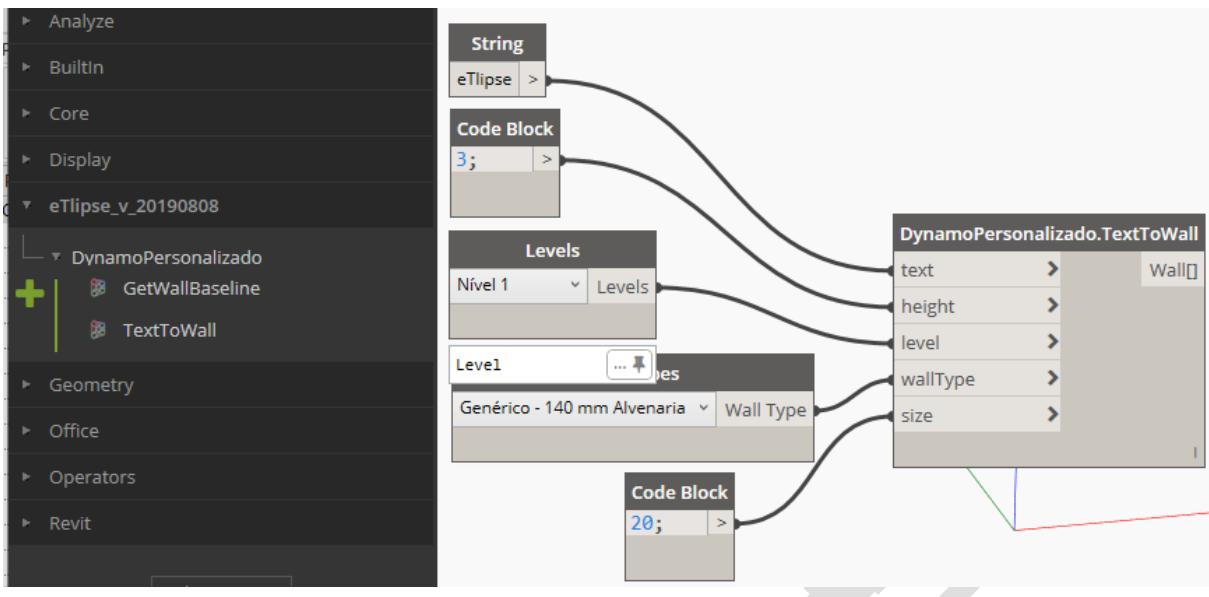
                    gp.AddString(text, font.FontFamily, (int)font.Style, font.Size, new
PointF(0, 0), sf);

                    // Converte pontos da fonte para pontos Dynamo
                    var points = gp.PathPoints.Select(p =>
Autodesk.DesignScript.Geometry.Point.ByCoordinates(p.X, -p.Y, 0)).ToList();
                    var types = gp.PathTypes;

                    Autodesk.DesignScript.Geometry.Point start = null;
                    // Cria linhas
                    for (var i = 0; i < types.Count(); i++) {
                        //Types:
                        //0 início da forma
                        //1 ponto em linha
                        //3 ponto em curva
                        //129 parcial final de linha
                        //131 parcial final de curva
                        //161 final de linha
                        //163 final de curva
                        if (types[i] == 0) {
                            start = points[i];
                        }
                        // Algumas letras precisam ser fechadas e outras não
                        if (types[i] > 100) {
                            if (!points[i].IsAlmostEqual(start)) {

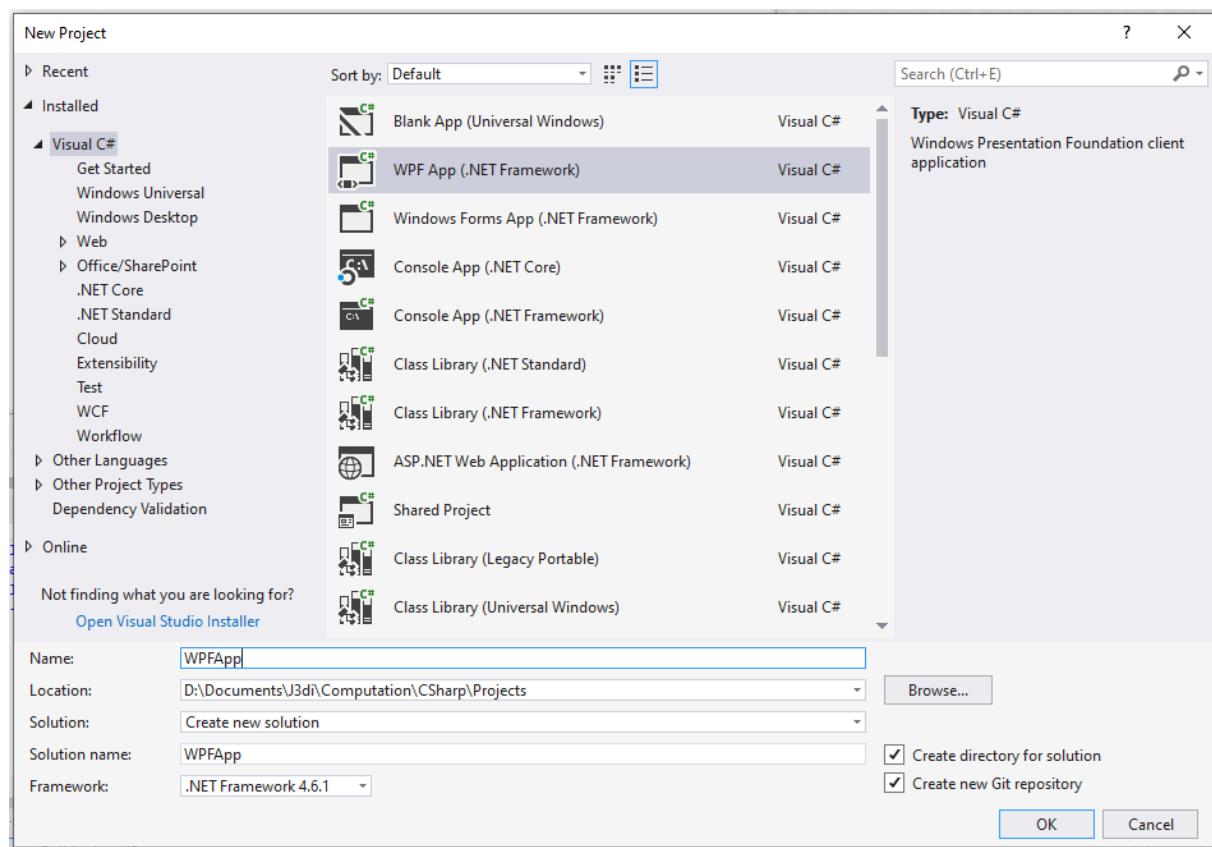
lines.Add(Autodesk.DesignScript.Geometry.Line.ByStartPointEndPoint(points[i], start));
                            }
                        } else {

lines.Add(Autodesk.DesignScript.Geometry.Line.ByStartPointEndPoint(points[i], points[i
+ 1]));
                        }
                    }
                    // Descartar pontos
                    foreach (var point in points) {
                        point.Dispose();
                    }
                    return lines;
                }
            }
        }
    }
```

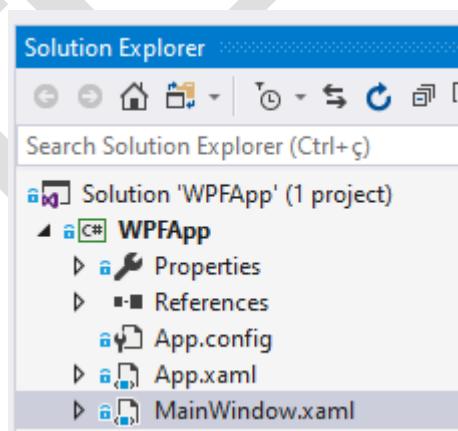


Exemplo - Usando WPF com Nodes Dynamo

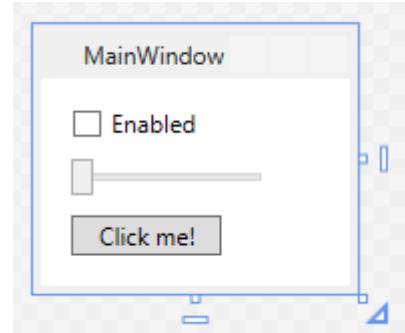
Adiante veremos como WPF, *Windows Presentation Foundation*, pode ser muito interessante, na verdade é uma importante recomendação usá-lo em lugar de Windows Forms. Os detalhes sobre WPF deixaremos para texto específico sobre Visual Studio. Iniciamos criando um aplicativo WPF no Visual Studio, de nome WPFApp.



Duplo clique no arquivo `MainWindow.xaml`, do Solution Explorer, para poder editar o arquivo. Usando a linguagem declarativa apropriada conseguimos adicionar os componentes apropriados.



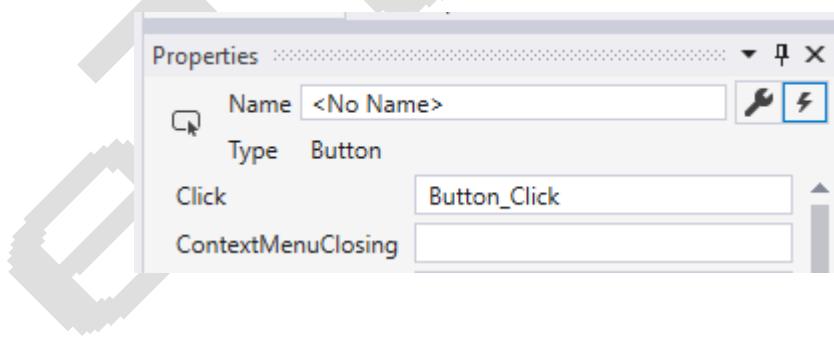
Nossa janela e seu respectivo código ficam como segue. Atenção que o botão terá seu evento Click programado.



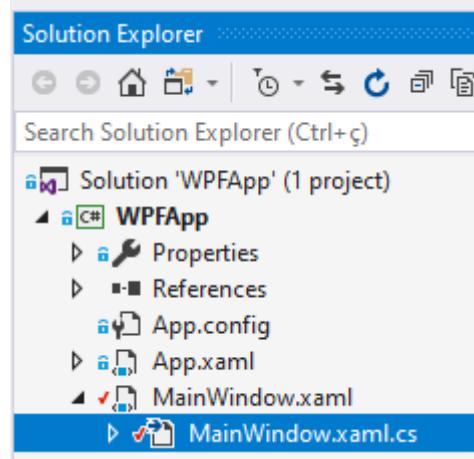
```

<Window x:Class="WPFApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WPFApp"
    mc:Ignorable="d"
    Title="MainWindow" SizeToContent="WidthAndHeight">
    <Grid Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <CheckBox Grid.Row="0" Margin="5" Name="EnabledCheckBox" Content="Enabled"
            HorizontalAlignment="Left" VerticalAlignment="Top"/>
        <Slider Grid.Row="1" Margin="5" Name="ValueSlider" HorizontalAlignment="Left"
            VerticalAlignment="Top" Width="100"/>
        <Button Grid.Row="2" Margin="5" Content="Click me!" HorizontalAlignment="Left"
            VerticalAlignment="Top" Width="75" Click="Button_Click"/>
    </Grid>
</Window>

```



A lógica do clique no botão deve ser definida no arquivo `MainWindow.xaml.cs`, que podemos localizar no Solution Explorer aninhado com o arquivo `MainWindow.xaml`. O código fica como segue.



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPFAApp {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e) {
            MessageBox.Show("Value is " + ValueSlider.Value);
        }
    }
}

```

O CheckBox será usando com o uso de Binding. Este recurso permite controlar uma propriedade associada a outro elemento, neste caso para habilitar ou não o botão, conforme a CheckBox esteja ou não ativa. O código fica como segue.

```

<Window x:Class="WPFAApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WPFAApp"
        mc:Ignorable="d"
        Title="MainWindow" SizeToContent="WidthAndHeight">

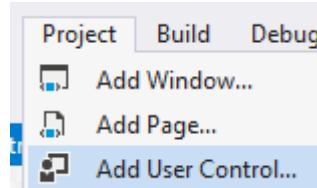
```

```

<Grid Margin="10">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <CheckBox Grid.Row="0" Margin="5" Name="EnabledCheckBox" Content="Enabled"
    HorizontalAlignment="Left" VerticalAlignment="Top"/>
    <Slider Grid.Row="1" Margin="5" Name="ValueSlider" HorizontalAlignment="Left"
    VerticalAlignment="Top" Width="100"/>
    <Button Grid.Row="2" Margin="5" Content="Click me!" HorizontalAlignment="Left"
    VerticalAlignment="Top" Width="75" Click="Button_Click"
    IsEnabled="{Binding ElementName=EnabledCheckBox, Path=IsChecked}"/>
</Grid>
</Window>

```

Podemos transformar nossos elementos em um único controle personalizado, o que nos permite um reuso mais fácil. Usamos o caminho Project>Add User Control e criamos um controle com nome MyCustomControl.xaml. Em seguida poderemos substituir nosso conjunto de controles pelo controle único personalizado que criamos.



Podemos substituir os controles dentro da tag <Grid>...</Grid> em MyCustomControl.xaml com as de MainWindow.xaml. Mesmo procedimento deve ser feito para a função Button_Click em MainWindow.xaml.cs and que moveremos para MyCustomControl.xaml.cs.

O procedimento cria controle personalizado reusável que pode ser aninhado dentro de outro controle WPF. Podemos agora adicionar o controle criado em MainWindow.xaml, o que pode ser feito adicionando o atributo xmlns:local="clr-namespace:WpfApp", e finalmente podemos declarar o controle personalizado, conforme código abaixo.

MyCustomControl.xaml

```

<UserControl x:Class="WpfApp.MyCustomControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    xmlns:local="clr-namespace:WpfApp"
    mc:Ignorable="d"
    d:DesignHeight="450" d:DesignWidth="800">

```

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <CheckBox Grid.Row="0" Margin="5" Name="EnabledCheckBox" Content="Enabled"
    HorizontalAlignment="Left" VerticalAlignment="Top"/>
    <Slider Grid.Row="1" Margin="5" Name="ValueSlider" HorizontalAlignment="Left"
    VerticalAlignment="Top" Width="100"/>
    <Button Grid.Row="2" Margin="5" Content="Click me!" HorizontalAlignment="Left"
    VerticalAlignment="Top" Width="75" Click="Button_Click"
    IsEnabled="{Binding ElementName=EnabledCheckBox, Path=IsChecked}"/>
</Grid>
</UserControl>

```

MyCustomControl.xaml.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPFAApp {
    /// <summary>
    /// Interaction logic for MyCustomControl.xaml
    /// </summary>
    public partial class MyCustomControl : UserControl {
        public MyCustomControl() {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e) {
            MessageBox.Show("Value is " + ValueSlider.Value);
        }
    }
}

```

MainWindow.xaml

```

<Window x:Class="WPFAApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WPFAApp"
    mc:Ignorable="d"
    Title="MainWindow" SizeToContent="WidthAndHeight">

```

```
<Grid Margin="10">
    <local:MyCustomControl/>
</Grid>
</Window>
```

MainWindow.xaml.cs

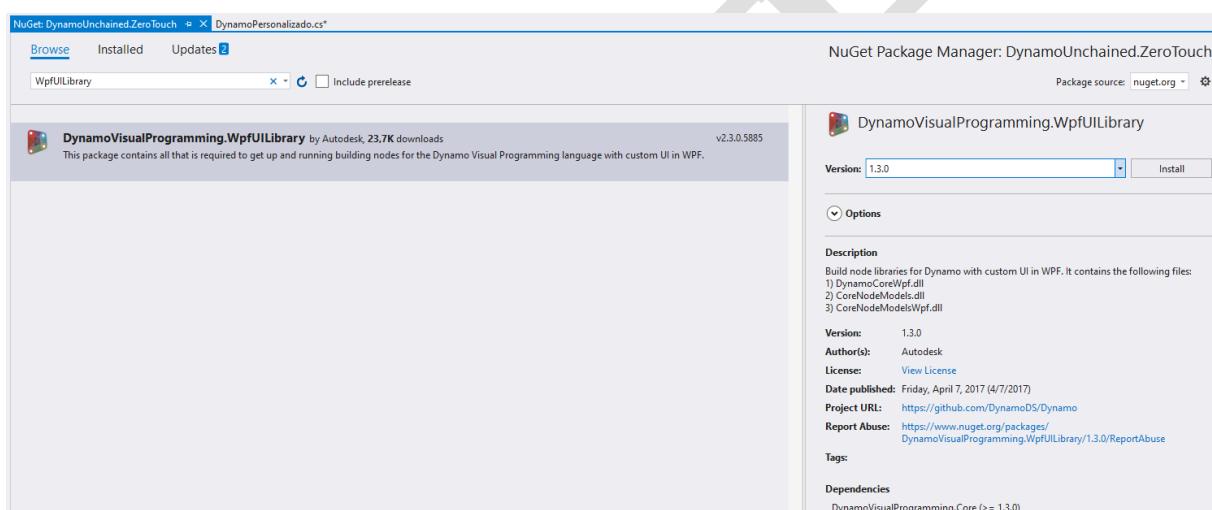
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WPFAApp {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
    }
}
```

Agora podemos avançar para personalizar um node do Dynamo com WPF. Precisaremos configurar um projeto no Visual Studio, tal como foi feito nos exemplos anteriores para personalizar para o Dynamo. Desta vez precisaremos acrescentar uma nova dependência, usando o NuGet, neste caso WpfUILibrary. Lembrar de configurar as bibliotecas adicionadas para não serem copiadas pro local, tal como feito anteriormente. Usaremos o nome de projeto DynamoUnchained.ExplicitNode, conforme sugerido no tutorial Zero Touch.

O arquivo json agora fica:

```
{
  "license": "",
  "file_hash": null,
  "name": "Dynamo Unchained - ExplicitNode",
  "version": "1.0.0",
  "description": "ExplicitNode sample node for the Dynamo Unchained workshop",
  "group": "",
  "keywords": null,
  "dependencies": [],
  "contents": "",
  "engine_version": "1.3.0.0",
  "engine": "dynamo",
  "engine_metadata": "",
  "site_url": "",
  "repository_url": "",
  "contains_binaries": true,
  "node_libraries": [
    "DynamoUnchained.ExplicitNode, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
  ]
}
```



Criaremos uma nova classe, HelloUI.cs, que implementa a interface NodeModel, algo que as classes nativas de nodes precisam fazer. As diretivas antes da classe definem a categoria no Dynamo, não sendo necessário o arquivo _DynamoCustomization.xml.

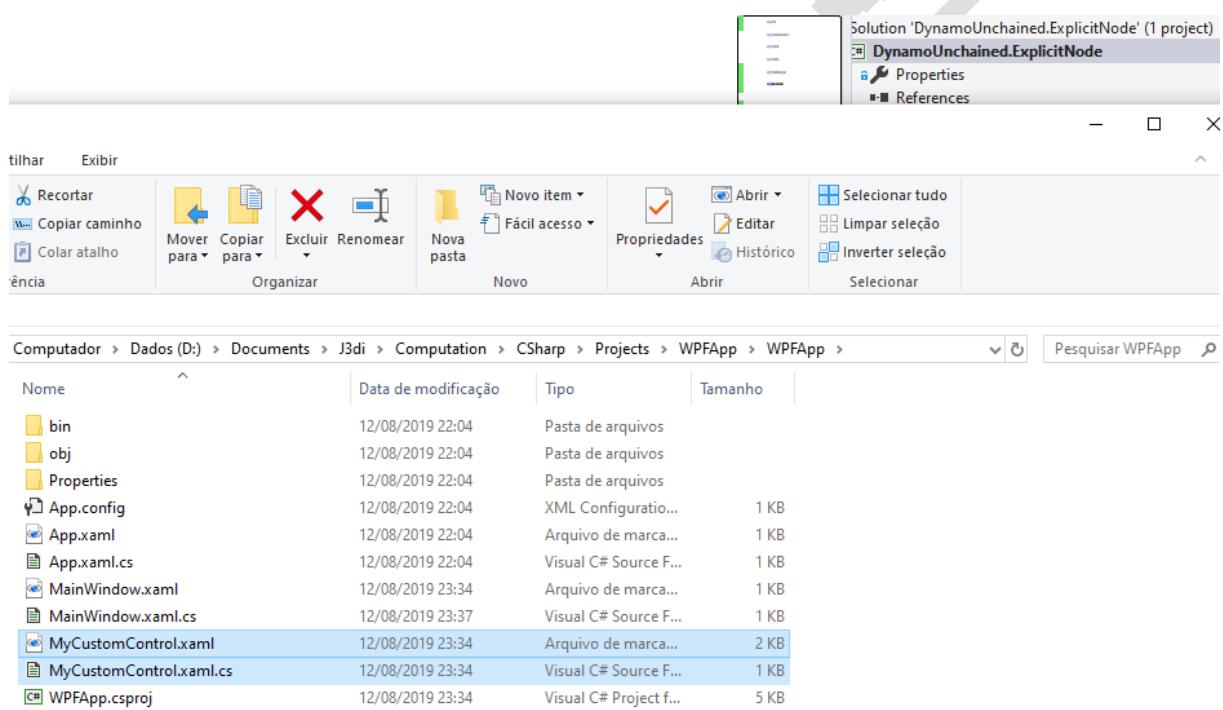
```

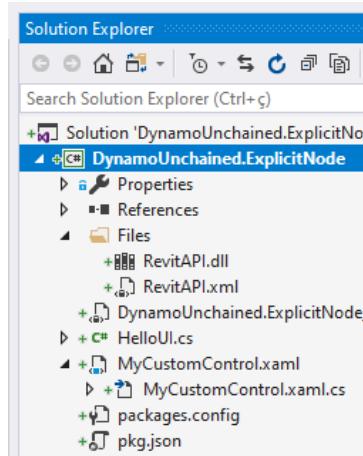
/* dynamo directives */
using Dynamo.Graph.Nodes;

namespace DynamoUnchained.ExplicitNode {
    [NodeName("HelloUI")]
    [NodeDescription("Sample Explicit Node")]
    [NodeCategory("Dynamo Unchained.Explicit Node")]
    [IsDesignScriptCompatible]
    public class HelloUI : NodeModel {
        public HelloUI() {
        }
    }
}

```

Faremos uso dos controles personalizados que criamos anteriormente, para isso precisamos incluir no atual projeto MyCustomControl.xaml e MyCustomControl.xaml.cs. Podemos incluir estes arquivos arrastando os arquivos para o projeto a partir do File Explorer. Precisamos atualizar o namespace nos dois arquivos para o que estamos usando no atual projeto, DynamoUnchained.ExplicitNode.



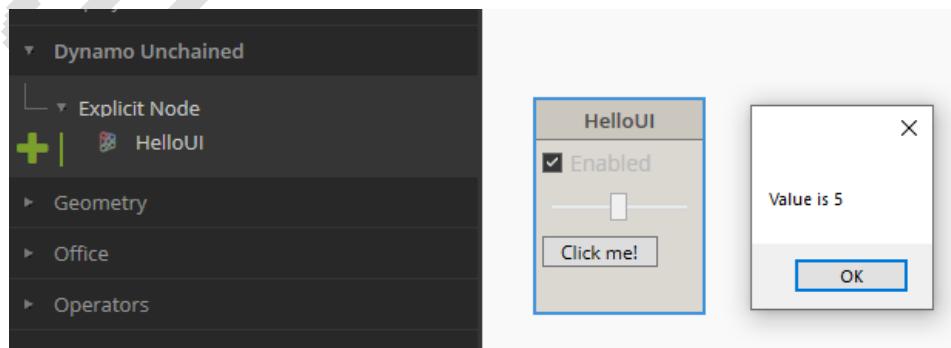


Precisamos criar uma classe que implemente a interface `INodeViewCustomization`, classe esta que iremos denominar `HelloUINodeView.cs`. Faremos a associação com o controle personalizado criado. Nossa código fica como segue e, após compilar, podemos fazer uso do node no Dynamo.

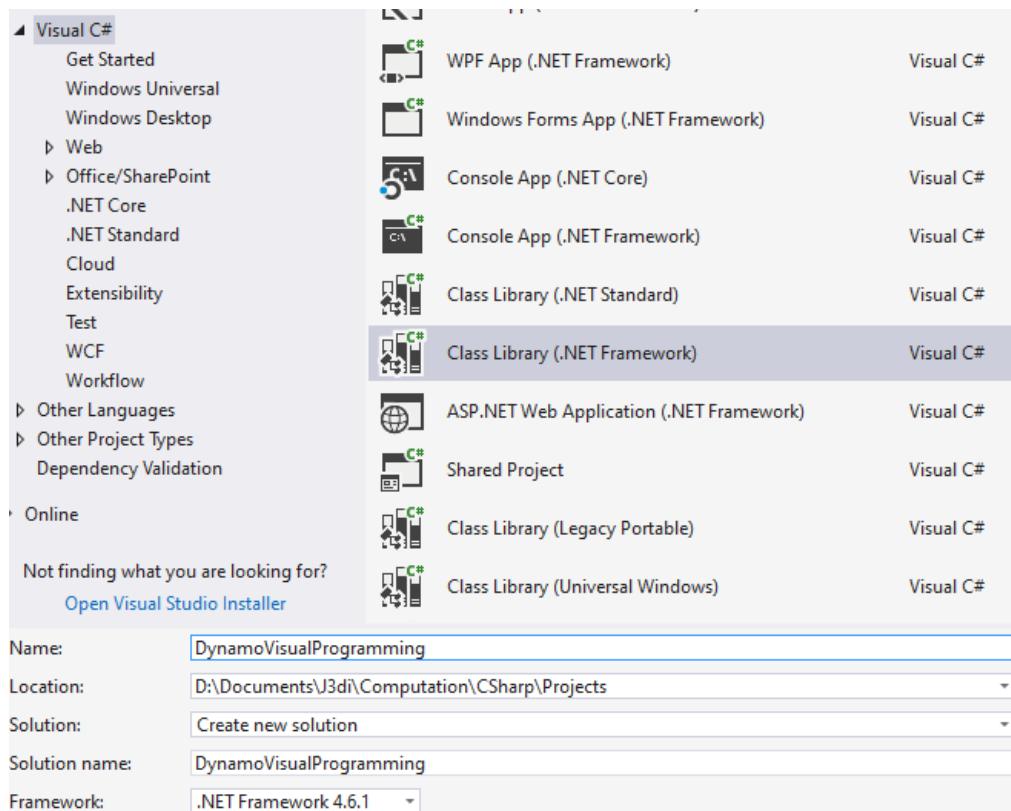
```
/* dynamo directives */
using Dynamo.Controls;
using Dynamo.Wpf;

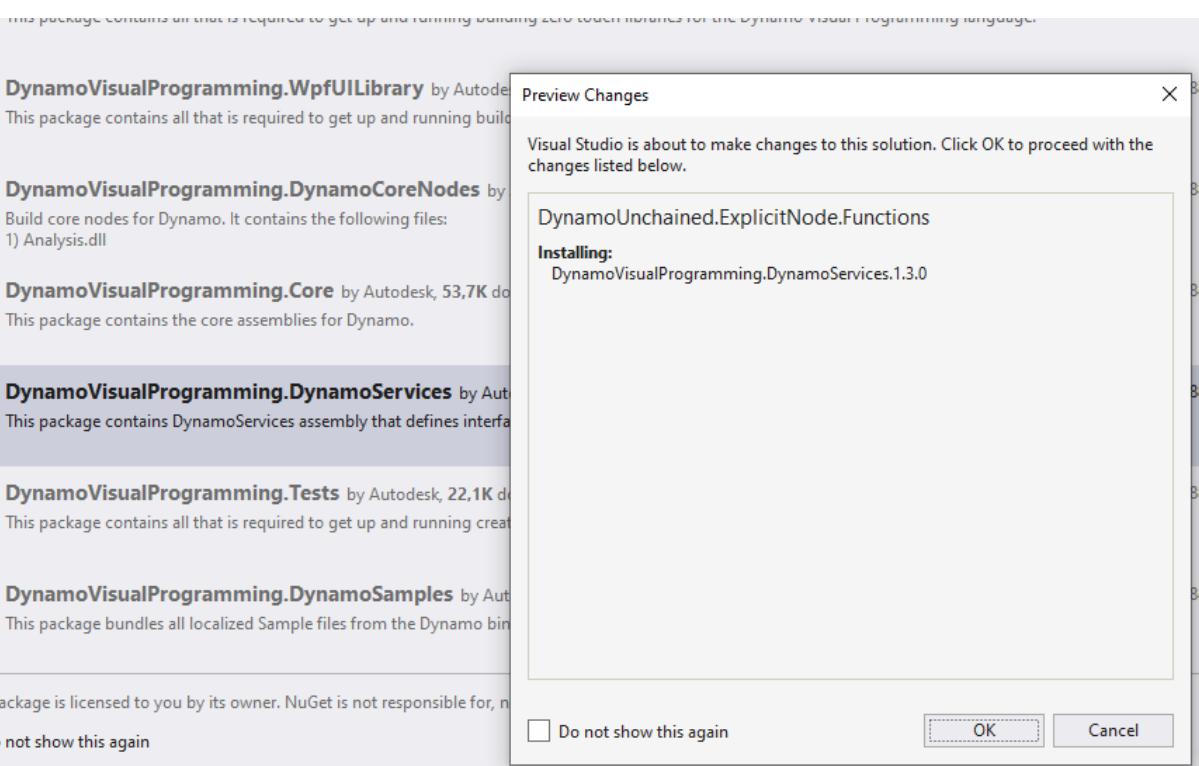
namespace DynamoUnchained.ExplicitNode {
    public class HelloUINodeView : INodeViewCustomization<HelloUI> {
        public void CustomizeView(HelloUI model, NodeView nodeView) {
            var ui = new MyCustomControl();
            nodeView.inputGrid.Children.Add(ui);
            ui.DataContext = model;
        }

        public void Dispose() {
        }
    }
}
```



Vamos avançar com a execução de função. Precisamos adicionar um projeto pois NodeModels executam um método BuildOutputAst que pega as entradas e passa para uma função que precisa estar em uma assembly separadamente. Vamos então adicionar um novo projeto DynamoUnchained.ExplicitNode.Functions. Em seguida devemos adicionar DynamoVisualProgramming.DynamoServices do pacote NuGet. Por fim adicionamos uma classe estática, Functions.cs, com o código conforme segue.



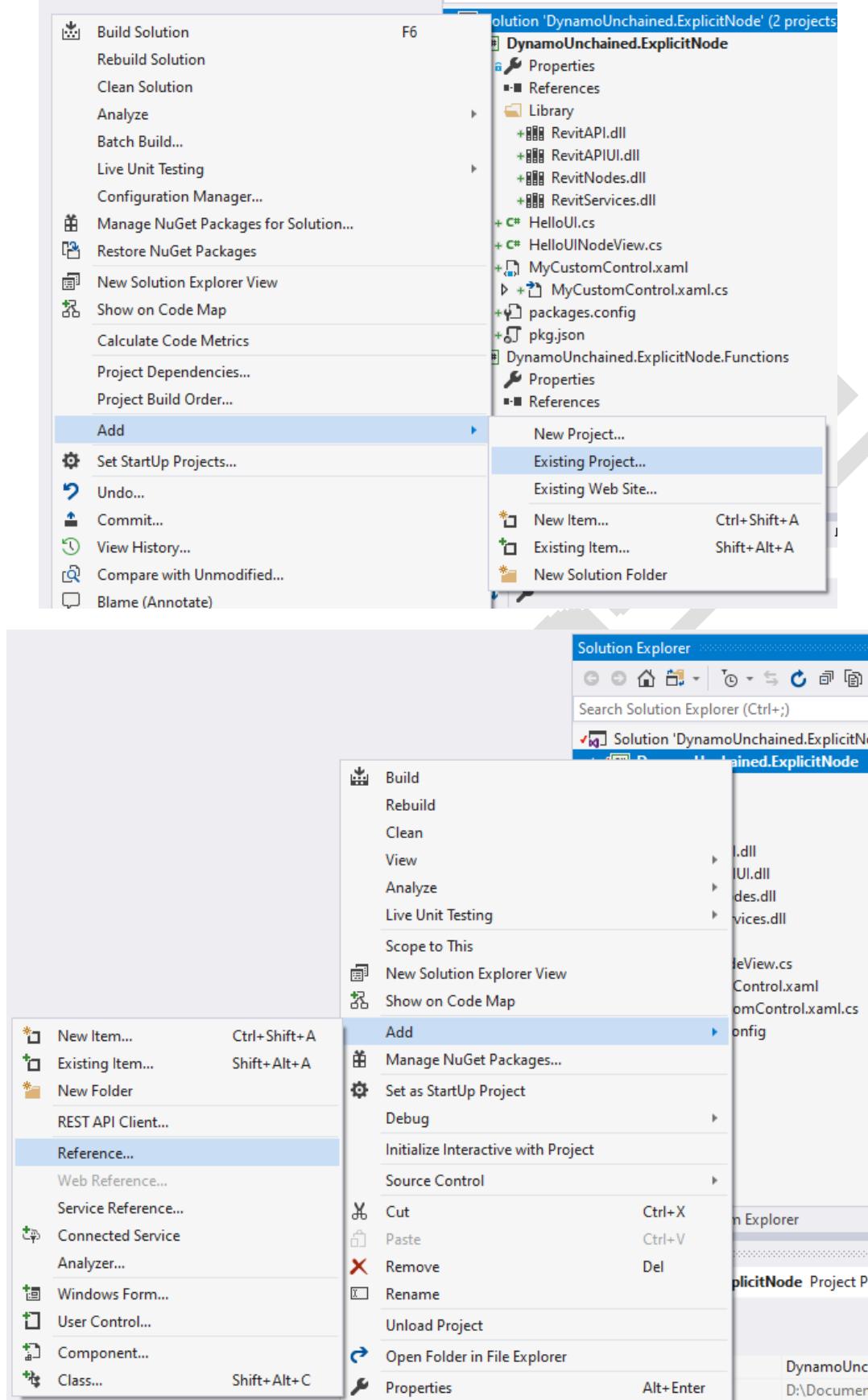


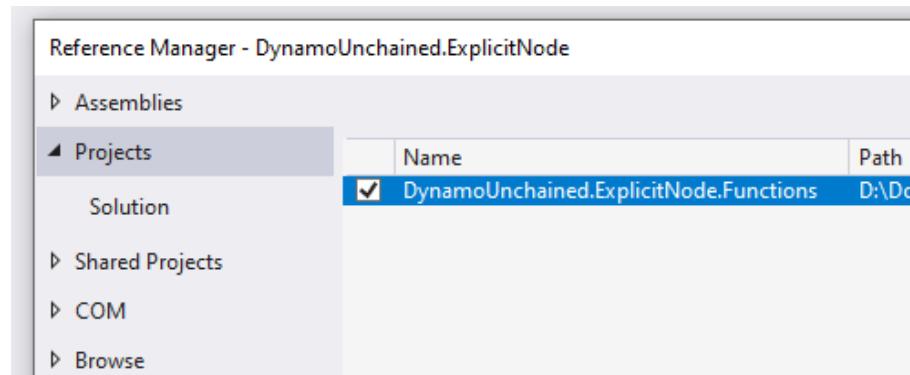
```

using Autodesk.DesignScript.Runtime;
namespace DynamoUnchained.ExplicitNode.Functions {
    [IsVisibleInDynamoLibrary(false)]
    public static class Functions {
        public static double MultiplyTwoNumbers(double a, double b) {
            return a * b;
        }
    }
}

```

Tudo pronto, agora podemos voltar para nosso projeto anterior e implementar `BuildOutputAst` dentro da classe `HelloUI.cs`. Devemos adicionar o projeto `DynamoUnchained.ExplicitNode.Functions` que criamos usando clique com o botão direito em `DynamoUnchained.ExplicitNode` e selecionando adicionar projeto.





O arquivo HelloUI.cs fica:

```

using System;
using System.Collections.Generic;
/* dynamo directives */
using Dynamo.Graph.Nodes;
using ProtoCore.AST.AssociativeAST;

namespace DynamoUnchained.ExplicitNode {
    [NodeName("HelloUI")]
    [NodeDescription("Sample Explicit Node")]
    [NodeCategory("DynamoUnchained")]
    [InPortNames("A")]
    [InPortTypes("double")]
    [InPortDescriptions("Number A")]
    [OutPortNames("Output")]
    [OutPortTypes("double")]
    [OutPortDescriptions("Product of two numbers")]
    [IsDesignScriptCompatible]
    public class HelloUI : NodeModel {
        public HelloUI() {
            RegisterAllPorts();
        }

        private double _sliderValue;

        public double SliderValue {
            get { return _sliderValue; }
            set {
                _sliderValue = value;
                RaisePropertyChanged("SliderValue");
                OnNodeModified(false);
            }
        }

        public override IEnumerable<AssociativeNode>
        BuildOutputAst(List<AssociativeNode> inputAstNodes) {
            if (!HasConnectedInput(0)) {
                return new[] {
                    AstFactory.BuildAssignment(GetAstIdentifierForOutputIndex(0),
                    AstFactory.BuildNullNode());
            }
            var sliderValue = AstFactory.BuildDoubleNode(SliderValue);
            var functionCall =
                AstFactory.BuildFunctionCall(

```

```

        new Func<double, double,
double>(Functions.Functions.MultiplyTwoNumbers),
        new List<AssociativeNode> { inputAstNodes[0], sliderValue });

    return new[] {
AstFactory.BuildAssignment(GetAstIdentifierForOutputIndex(0), functionCall) };
}
}
}

```

O arquivo MyCustomControl.XAML fica:

```

<UserControl x:Class="DynamoUnchained.ExplicitNode.MyCustomControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    xmlns:local="clr-namespace:DynamoUnchained.ExplicitNode"
    mc:Ignorable="d"
    d:DesignHeight="450" d:DesignWidth="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <CheckBox Grid.Row="0" Margin="5" Name="EnabledCheckBox" Content="Enabled"
HorizontalAlignment="Left" VerticalAlignment="Top"/>
        <Slider
            Name="ValueSlider"
            Grid.Row="1"
            Width="100"
            Margin="5"
            HorizontalAlignment="Left"
            VerticalAlignment="Top"
            IsSnapToTickEnabled="True"
            TickFrequency="1"
            Value="{Binding SliderValue}"/>
        <Button Grid.Row="2" Margin="5" Content="Click me!" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="75" Click="Button_Click"
            IsEnabled="{Binding ElementName=EnabledCheckBox, Path=IsChecked}"/>
    </Grid>
</UserControl>

```

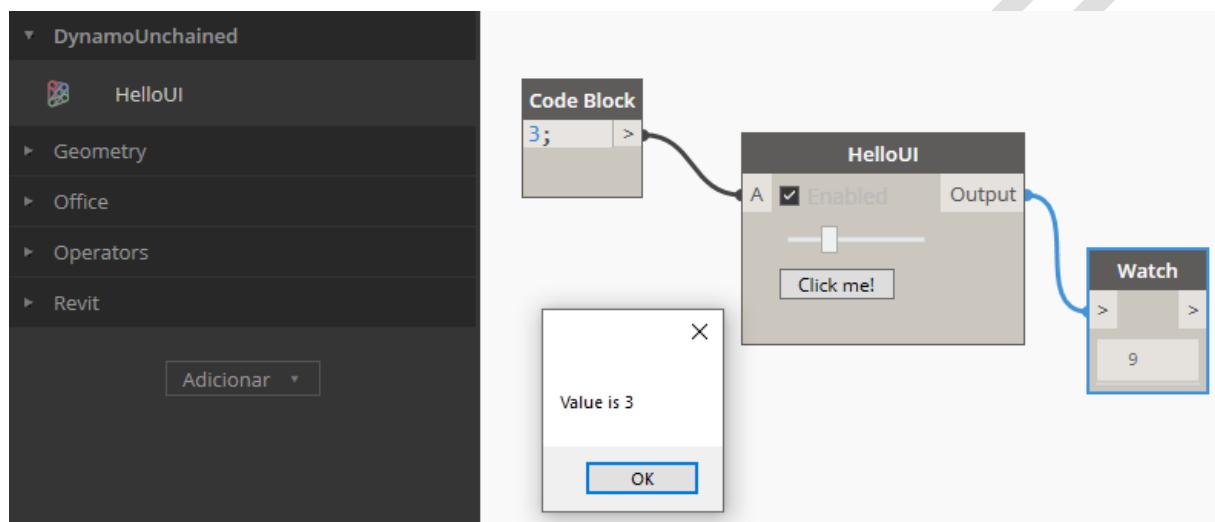
Por fim editamos o arquivo pkg.json para orientar o Dynamo a carregar DynamoUnchained.ExplicitNode.Functions.dll. Em seguida poderemos compilar e fazer o teste do node criado. O node OnNodeModified() informa o Dynamo quando um nó é modificado.

```
{
    "license": "",
    "file_hash": null,
    "name": "Dynamo Unchained - ExplicitNode",
    "version": "1.0.0",
    "description": "ExplicitNode sample node for the Dynamo Unchained workshop",
    "group": "",
    "keywords": null,
}
```

```

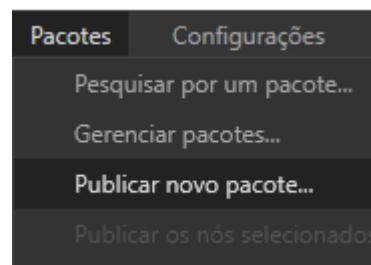
"dependencies": [],
"contents": "",
"engine_version": "1.3.0.0",
"engine": "dynamo",
"engine_metadata": "",
"site_url": "",
"repository_url": "",
"contains_binaries": true,
"node_libraries": [
    "DynamoUnchained.ExplicitNode, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null",
    "DynamoUnchained.ExplicitNode.Functions, Version=1.0.0.0, Culture=neutral,
    PublicKeyToken=null"
]
}

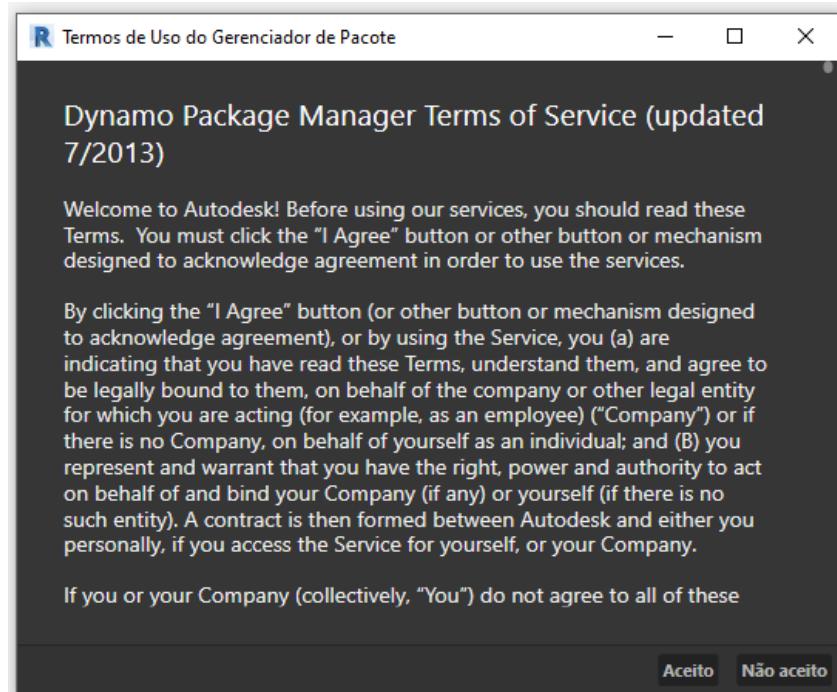
```



Publicando pacotes no Dynamo para o Package Manager

Uma vez que um novo pacote foi desenvolvido, e adequadamente testado para não conter erros, é possível publicá-lo. O Dynamo possuí estavas funcionalidade, conforme figura que segue.

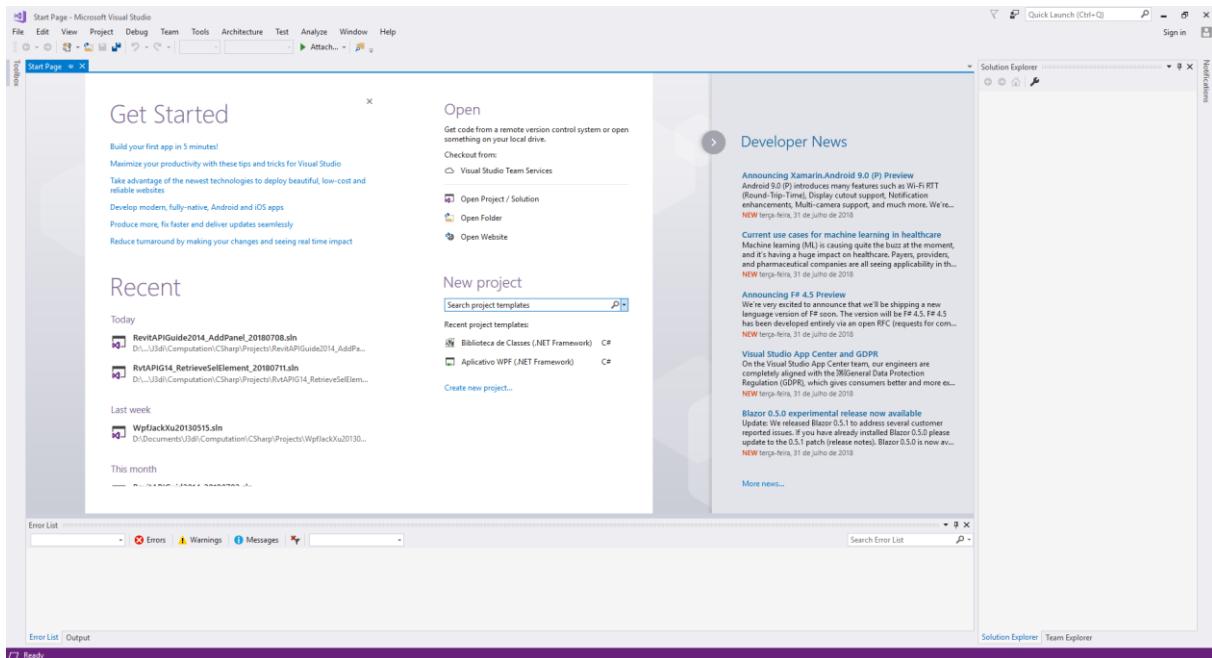




Tudo certo com a publicação, o pacote ficará disponível para download. O Dynamo traz a possibilidade de utilizar pacotes desenvolvidos por outros programadores ou comunidades.

Visual Studio

Plataforma para desenvolvimento de serviços e aplicativos computacionais.



A plataforma Visual Studio da Microsoft consiste em um ambiente de desenvolvimento integrado (IDE) onde é possível o desenvolvimento de software. É específico para o .NET Framework, onde é possível usar as linguagens Visual Basic (VB), C, C++, C# (C Sharp) e F# (F Sharp), além de outras soluções, como o desenvolvimento na área web, usando a plataforma do ASP.NET, como websites, aplicativos web, serviços web e aplicativos móveis. As linguagens que são mais usadas no Visual Studio são o VB.NET, Visual Basic.Net, e o C#, cuja pronúncia é C Sharp.

WPF - Windows Presentation Foundation.

Uma peça muito importante em um aplicativo é sua interface. A interface é responsável por captar a entrada de dados do usuário e, após certo processamento, apresentar os resultados adequadamente. O Windows substituiu sua interface padrão, de Windows Form, que era baseado em imagens raster, matriz de pontos. O WPF, Windows Presentation Foundation, é baseado em uma

apresentação vetorial, resultando em imagens melhor definidas e apresentadas, sendo mais leve computacionalmente, dentro outros benefícios, como não distorções.

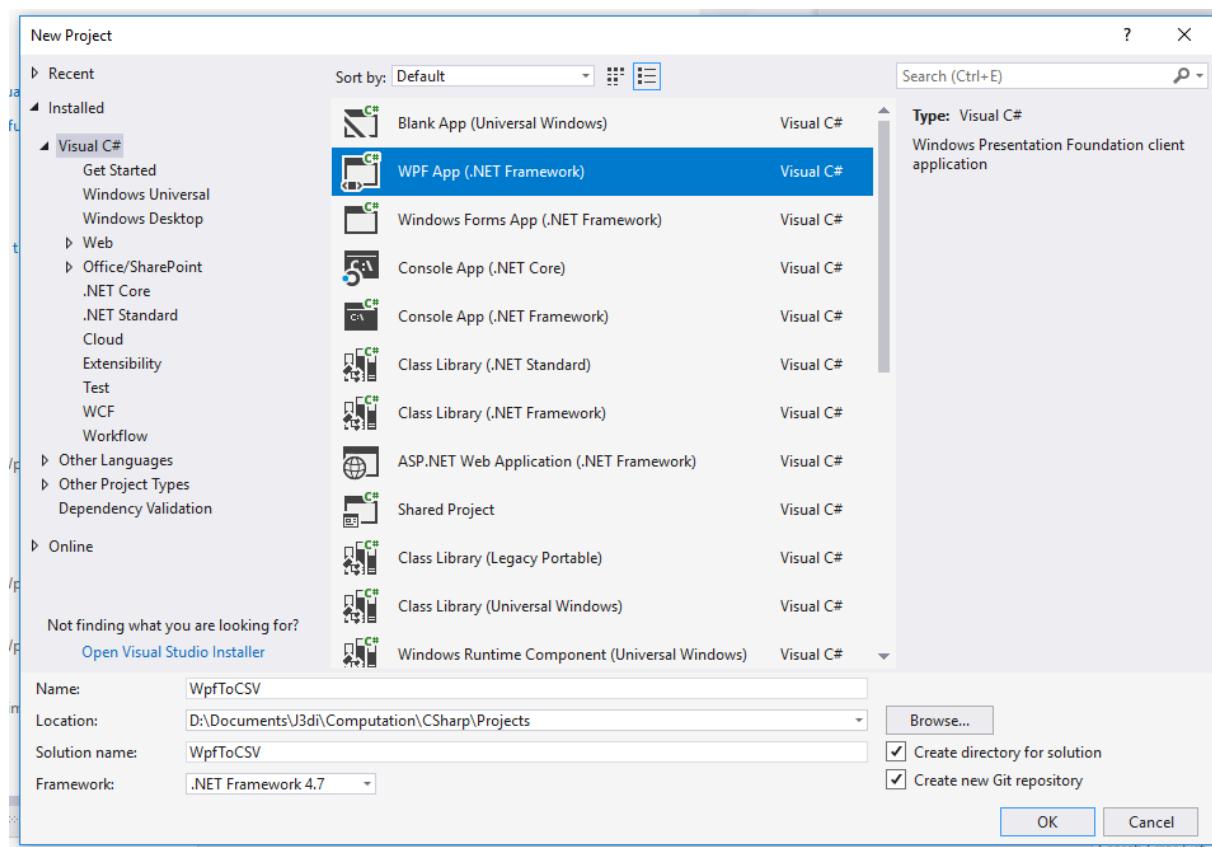
A forma de definir uma interface se dá pelo uso do XAML, pronunciado zammel em inglês ou zimel em português, e que significa eXtensible Application Markup Language. XAML é uma linguagem declarativa baseada no XML e possui suporte nativo para edição no Visual Studio. A interface pode ser definida arrastando-se os componentes para a janela padrão, ou pela edição textual do XAML.

Exemplo - Projeto WPF para Gerar CSV

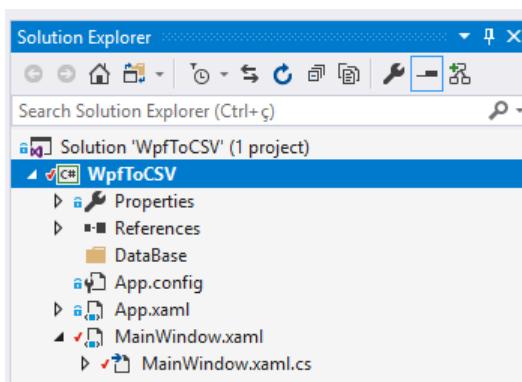
Através deste exemplo entenderemos melhor o uso desta tecnologia. Abriremos o Visual Studio e iniciaremos um novo projeto.

- Menu File
- New
- Project

As opções de projeto são apresentadas. Devemos optar por WPF App e optar pelo nome “WpfToCSV”.



No Solution Explorer podemos ver os arquivos criados, onde também podemos adicionar pastas para melhor organização. O arquivo “MainWindow.xaml” contém a interface do aplicativo. O arquivo “MainWindow.cs” é o arquivo da lógica do programa, que neste caso será escrita em C#. Duplo clique nestes arquivos possibilita a abertura para edição.



O foco deste trabalho não é detalhar o WPF, portanto, apresentamos o código completo sem entrar em detalhes sobre sua sintaxe. É interessante iniciar pela interface e depois promover a lógica, usando os componentes da interface. Isto dito para a codificação quando já estamos com o projeto pronto, algo que normalmente fazemos com UML.

Código XAML:

```

<Window x:Class="WpfToCSV.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfToCSV"
        mc:Ignorable="d"
        Title="Creating CSV" Height="170" Width="330">
    <StackPanel>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>
            <TextBlock Text="Nível" FontWeight="Bold" Margin="5" Grid.Column="1"
Grid.Row="0"/>
            <TextBlock Text="Material" FontWeight="Bold" Margin="5" Grid.Column="2"
Grid.Row="0"/>
            <!-->
            <TextBlock Text="Janela:" FontWeight="Bold" Margin="5" Grid.Column="0"
Grid.Row="1"/>
            <TextBlock Text="Porta:" FontWeight="Bold" Margin="5" Grid.Column="0"
Grid.Row="2"/>
            <!-->
            <TextBox Name="txtJanelaNivel" Width="100" Margin="5" Grid.Column="1"
Grid.Row="1"/>
            <TextBox Name="txtJanelaMaterial" Width="100" Margin="5" Grid.Column="2"
Grid.Row="1"/>
            <!-->
            <TextBox Name="txtPortaNivel" Width="100" Margin="5" Grid.Column="1"
Grid.Row="2"/>
            <TextBox Name="txtPortaMaterial" Width="100" Margin="5" Grid.Column="2"
Grid.Row="2"/>
        </Grid>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <Button Name="btnLimpar" Width="51" Margin="5"
Click="BtnLimpar_Click">_Limpar</Button>
            <Button Name="btnCSV" Width="51" Margin="5"
Click="BtnCSV_Click">_CSV</Button>
            <Button Name="btnExit" Width="51" Margin="5,5,37,5"
Click="BtnExit_Click">Sai_r</Button>
        </StackPanel>
    
```

```

        </StackPanel>
</Window>
```

Código C#:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.IO;
using Microsoft.Win32;

namespace WpfToCSV {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window {

        // Variáveis globais:
        string arquivoNome;
        public string ArquivoName { get { return arquivoNome; } set { arquivoNome = value; } }

        public MainWindow() {
            InitializeComponent();
            ArquivoName = "";
        }

        private void BtnExit_Click(object sender, RoutedEventArgs e) {
            this.Close();
        }

        private void BtnLimpar_Click(object sender, RoutedEventArgs e) {
            txtJanelaMaterial.Text = "";
            txtJanelaNivel.Text = "";
            txtPortaMaterial.Text = "";
            txtPortaNivel.Text = "";
            // Tip: to get value as number var = Convert.ToDouble(txtField.Text)
        }

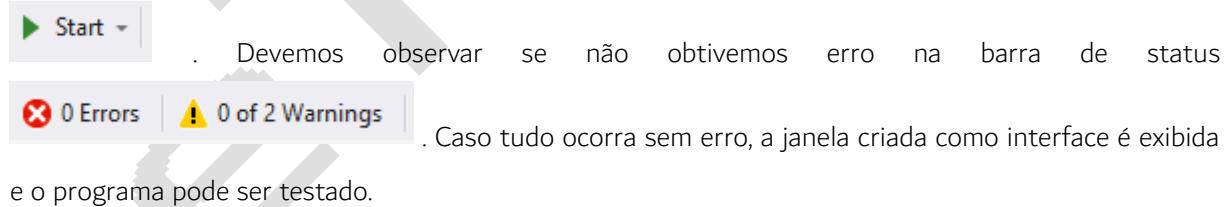
        private void BtnCSV_Click(object sender, RoutedEventArgs e) {
            try {
                string CombinedPath =
System.IO.Path.Combine(Directory.GetCurrentDirectory(), "..\\..\\\\ DataBase");
                SaveFileDialog saveFileDialog = new SaveFileDialog {
                    Title = "Salvar CSV",
                    //Alternativas
```

```

        //InitialDirectory = @"..\..\DataBase\",
        //InitialDirectory = @"D\Temp\",
        //InitialDirectory = "D\\Temp\\",
        //InitialDirectory =
System.IO.Path.Combine(System.IO.Path.GetDirectoryName(Directory.GetCurrentDirectory()),
), "..\\..\..\DataBase\\"),
                InitialDirectory = System.IO.Path.GetFullPath(CombinedPath),
                Filter = "Rtf documents|*.rtf|Txt files (*.txt)|*.txt|Csv files
(*.csv)|*.csv|All files (*.*)|*.*",
                FilterIndex = 3,
                RestoreDirectory = true
            };
            saveFileDialog.ShowDialog();
            if (saveFileDialog.FileName == "") {
                MessageBox.Show("Arquivo Inválido", "Salvar Como",
MessageBoxButton.OK);
            } else {
                ArquivoName = saveFileDialog.FileName;
                using (StreamWriter writer = new StreamWriter(ArquivoName)) {
                    writer.WriteLine("Elemento;Nível;Material");
                    writer.WriteLine("Porta:{};{}", txtPortaNivel.Text,
txtPortaMaterial.Text);
                    writer.WriteLine("Janela:{};{}", txtJanelaNivel.Text,
txtJanelaMaterial.Text);
                }
                MessageBox.Show("Arquivo Salvo com Sucesso!", "Salvar!",
MessageBoxButton.OK);
            }
        } catch (Exception ex) {
            MessageBox.Show(ex.ToString());
        }
    }
}
}

```

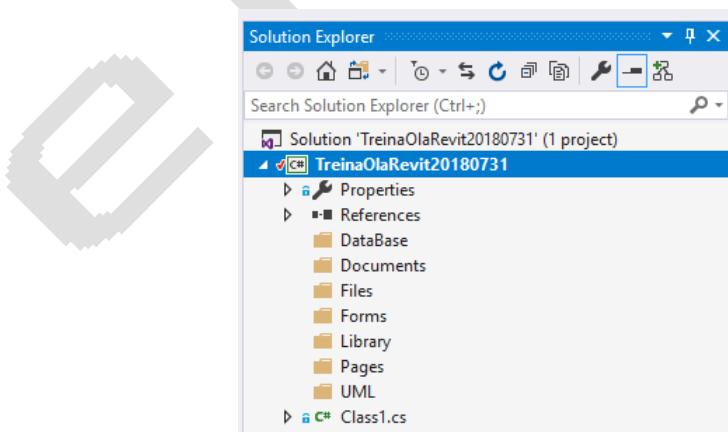
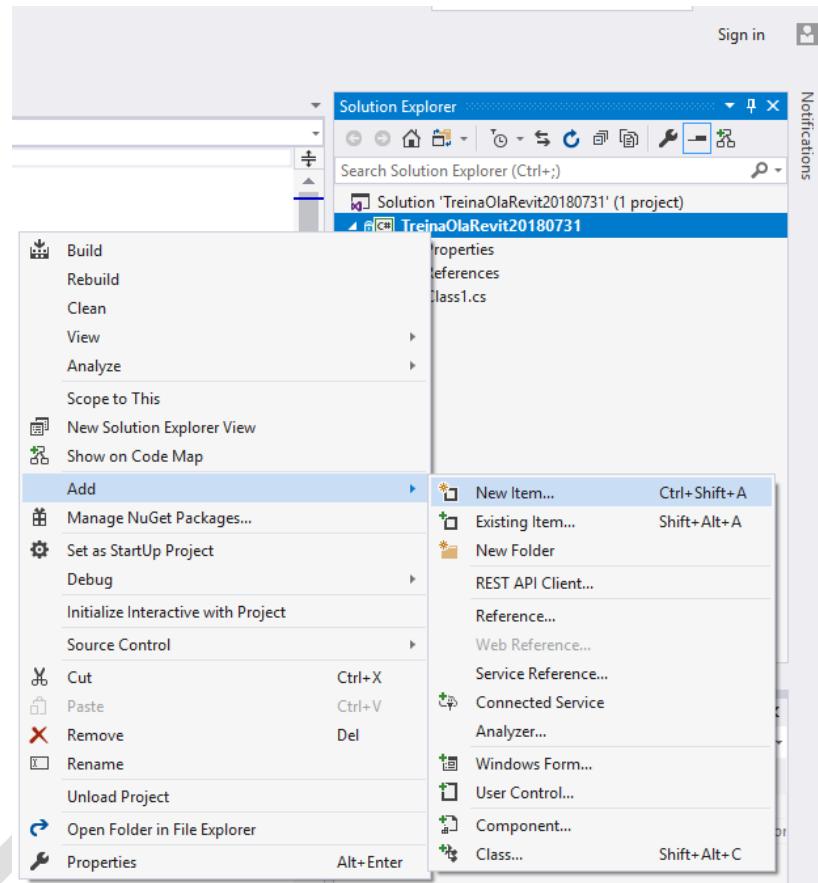
O programa deve ser copilado para que um executável seja gerado, caso não seja encontrado erro grave. O botão para compilação, na parte superior sob o menu, é uma das formas para compilar o arquivo



Boas práticas:

```
/*
```

Criar pastas no Solution Explorer pode ajudar a melhor organizar os arquivos do projeto. Esta possibilidade é possível com um clique do botão direito do mouse no projeto criado, na janela Solution Explorer.



```
*/
```

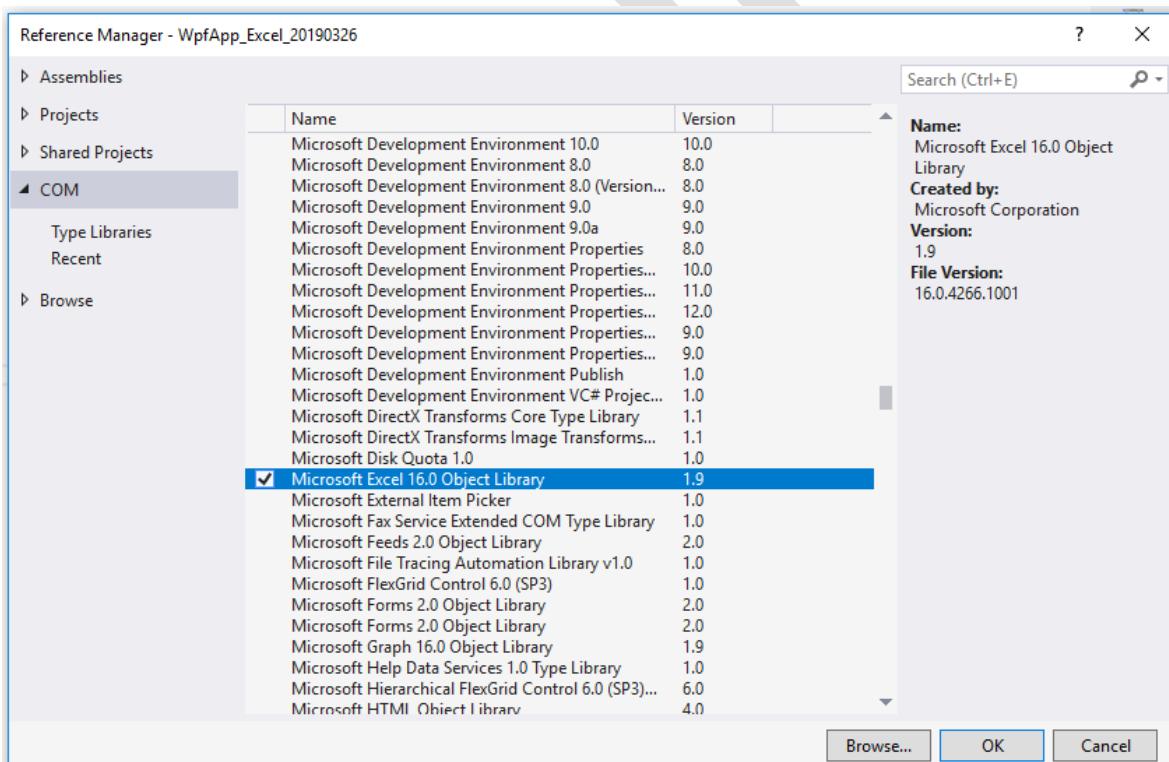
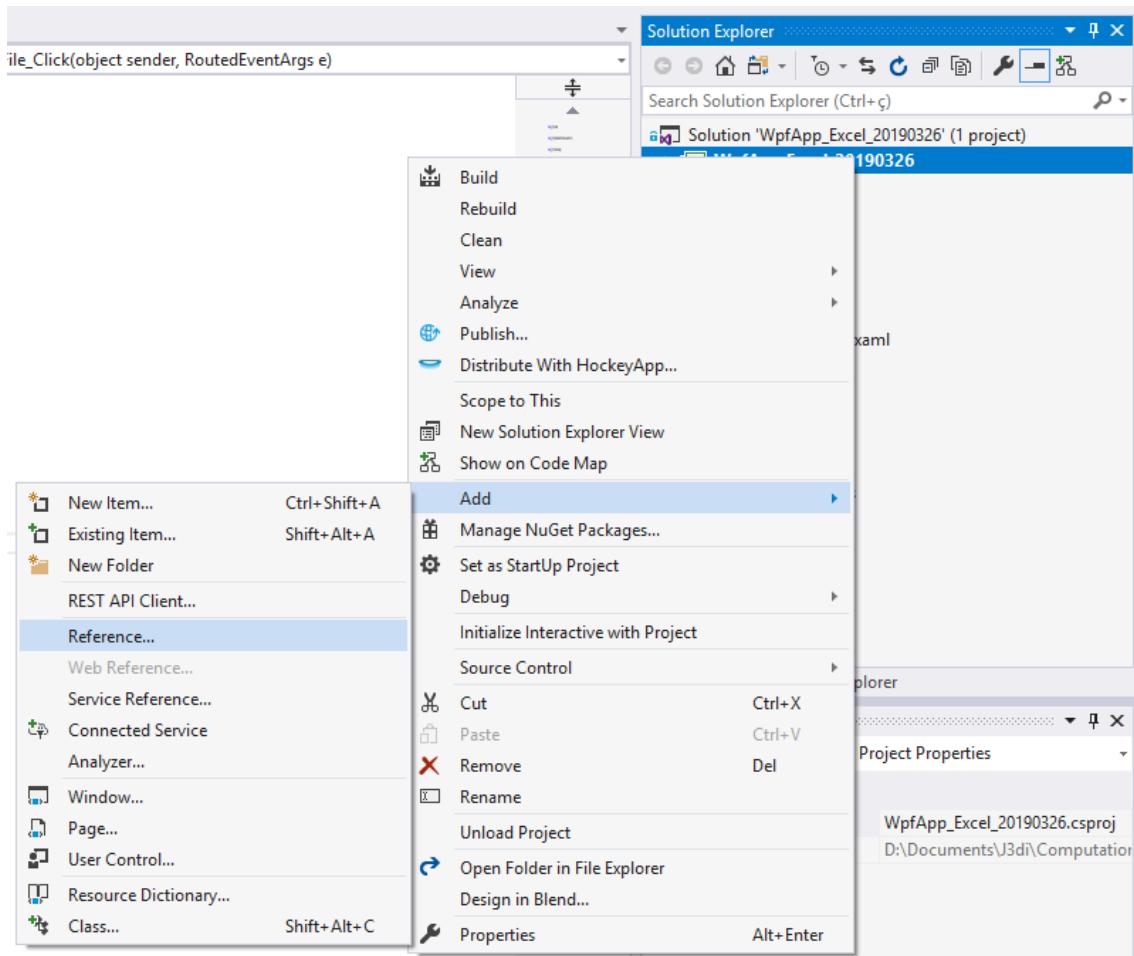
Utilizando o Excel como Base de Dados

Obviamente existem diversas opções de banco de dados, existindo soluções escaláveis que propiciam recursos mais apropriados para aplicações de maior porte, tais como escalabilidade e segurança.

Iremos exemplificar o uso do Excel como alternativa apenas para facilitar o estudo inicial, além de permitir uma solução que normalmente é mais encontrada em computadores pessoais. Através deste exemplo entenderemos como realizar a conexão com o Excel, que pode ser uma alternativa simplificada interessante como base de dados.

Exemplo - Interagindo com o Excel

Abriremos o Visual Studio e iniciaremos um novo projeto WPF. Precisamos adicionar a referência apropriada para ter acesso ao Excel.



Uma vez que este não é o foco neste material, apresentamos o exemplo sem detalhar passo a passo, embora o código esteja separado de forma que facilita o entendimento. O projeto ficou organizado como segue.

Arquivo da interface FrmStartUp.xaml.

```
<Window x:Class="WpfApp_Excel_20190326.Forms.FrmStartUp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfApp_Excel_20190326.Forms"
    mc:Ignorable="d"
    Title="Exemplos de Interação com o Excel" Height="150" Width="450">
    <StackPanel>
        <!-- Main Menu -->
        <Menu Height="21">
            <MenuItem Header="_File">
                <MenuItem Name="mnuExit" Header="E_xit" Click="Exit"/>
            </MenuItem>
        </Menu>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right"
Margin="5,33,5,5">
            <Button Name="btnAddPicture" Width="70" Margin="2,9,3,5"
Click="ReadAddPicture_Click">Add Picture</Button>
            <Button Name="btnAddSheet" Width="70" Margin="2,9,3,5"
Click="ReadAddSheet_Click">Add Sheet</Button>
            <Button Name="btnReadAllFile" Width="70" Margin="2,9,3,5"
Click="ReadAllFile_Click">Read All File</Button>
            <Button Name="btnOpenFile" Width="70" Margin="2,9,3,5"
Click="OpenFile_Click">Open File</Button>
            <Button Name="btnCreateFile" Width="60" Margin="2,9,3,5"
Click="CreateFile_Click">Create File</Button>
            <Button Name="btnExit" Width="50" Margin="2,9,3,5"
Click="Exit">E_xit</Button>
        </StackPanel>
    </StackPanel>
</Window>
```

Arquivo cs da janela principal FrmStartUp.xaml.cs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Runtime.InteropServices;
```

```

using Excel = Microsoft.Office.Interop.Excel;

namespace WpfApp_Excel_20190326.Forms {
    /// <summary>
    /// Interaction logic for FrmStartUp.xaml
    /// </summary>
    public partial class FrmStartUp : Window {
        public FrmStartUp() {
            InitializeComponent();
        }

        private void Exit(object sender, RoutedEventArgs e) {
            this.Close();
        }

        private void CreateFile_Click(object sender, RoutedEventArgs e) {
            Excel.Application xlApp = new
Microsoft.Office.Interop.Excel.Application();

            if (xlApp == null) {
                MessageBox.Show("Excel is not properly installed!!!");
                return;
            }
            Excel.Workbook xlWorkBook;
            Excel.Worksheet xlWorkSheet;
            object misValue = System.Reflection.Missing.Value;

            xlWorkBook = xlApp.Workbooks.Add(misValue);
            xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

            xlWorkSheet.Cells[1, 1] = "ID";
            xlWorkSheet.Cells[1, 2] = "Name";
            xlWorkSheet.Cells[2, 1] = "1";
            xlWorkSheet.Cells[2, 2] = "One";
            xlWorkSheet.Cells[3, 1] = "2";
            xlWorkSheet.Cells[3, 2] = "Two";

            xlWorkBook.SaveAs("d:\\temp\\csharp-Excel.xls",
Excel.XlFileFormat.xlWorkbookNormal, misValue, misValue, misValue, misValue,
Excel.XlSaveAsAccessMode.xlExclusive, misValue, misValue,
misValue, misValue, misValue);
            xlWorkBook.Close(true, misValue, misValue);
            xlApp.Quit();

            Marshal.ReleaseComObject(xlWorkSheet);
            Marshal.ReleaseComObject(xlWorkBook);
            Marshal.ReleaseComObject(xlApp);

            MessageBox.Show("Excel file created, you can find the file
d:\\temp\\csharp-Excel.xls");
        }

        private void OpenFile_Click(object sender, RoutedEventArgs e) {
            Excel.Application xlApp;
            Excel.Workbook xlWorkBook;
            Excel.Worksheet xlWorkSheet;
            object misValue = System.Reflection.Missing.Value;

            xlApp = new Excel.Application();
            xlWorkBook = xlApp.Workbooks.Open("d:\\temp\\csharp-Excel.xls", 0, true,
5, "", "", true, Microsoft.Office.Interop.Excel.XlPlatform.xlWindows, "\t",
false, false, 0, true, 1, 0);
        }
    }
}

```

```

xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

MessageBox.Show(xlWorkSheet.get_Range("A1", "A1").Value2.ToString());

xlWorkBook.Close(true, misValue, misValue);
xlApp.Quit();

releaseObject(xlWorkSheet);
releaseObject(xlWorkBook);
releaseObject(xlApp);
}

private void releaseObject(object obj) {
    try {
        System.Runtime.InteropServices.Marshal.ReleaseComObject(obj);
        obj = null;
    } catch (Exception ex) {
        obj = null;
        MessageBox.Show("Unable to release the Object " + ex.ToString());
    } finally {
        GC.Collect();
    }
}

private void ReadAllFile_Click(object sender, RoutedEventArgs e) {
    Excel.Application xlApp;
    Excel.Workbook xlWorkBook;
    Excel.Worksheet xlWorkSheet;
    Excel.Range range;

    string str;
    int rCnt;
    int cCnt;
    int rw = 0;
    int cl = 0;

    xlApp = new Excel.Application();
    xlWorkBook = xlApp.Workbooks.Open("d:\\temp\\csharp-Excel.xls", 0, true,
5, "", "", true, Microsoft.Office.Interop.Excel.XlPlatform.xlWindows, "\t", false,
false, 0, true, 1, 0);
    xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

    range = xlWorkSheet.UsedRange;
    rw = range.Rows.Count;
    cl = range.Columns.Count;

    for (rCnt = 1; rCnt <= rw; rCnt++) {
        for (cCnt = 1; cCnt <= cl; cCnt++) {
            // Error if not string in Excel!
            str = (string)(range.Cells[rCnt, cCnt] as Excel.Range).Value2;
            MessageBox.Show(str);
        }
    }

    xlWorkBook.Close(true, null, null);
    xlApp.Quit();

    Marshal.ReleaseComObject(xlWorkSheet);
    Marshal.ReleaseComObject(xlWorkBook);
    Marshal.ReleaseComObject(xlApp);
}
}

```

```

private void ReadAddSheet_Click(object sender, RoutedEventArgs e) {
    Excel.Application xlApp = new
Microsoft.Office.Interop.Excel.Application();

    if (xlApp == null) {
        MessageBox.Show("Excel is not properly installed!");
        return;
    }

    // Disable Excel overwrite prompt. Suppress prompts and alert messages
    // while a macro is running.
    xlApp.DisplayAlerts = false;

    string filePath = "d:\\temp\\csharp-Excel.xls";
    Excel.Workbook xlWorkBook =
xlApp.Workbooks.Open(filePath, 0, false, 5, "", "", false, Microsoft.Office.Interop.Excel.XlPlatform.xlWindows, "", true, false, 0, true, false, false);
    Excel.Sheets worksheets = xlWorkBook.Worksheets;

    var xlNewSheet = (Excel.Worksheet)worksheets.Add(worksheets[1],
Type.Missing, Type.Missing, Type.Missing);
    xlNewSheet.Name = "newsheet";
    xlNewSheet.Cells[1, 1] = "New sheet content";

    xlNewSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);
    xlNewSheet.Select();

    xlWorkBook.Save();
    xlWorkBook.Close();

    releaseObject(xlNewSheet);
    releaseObject(worksheets);
    releaseObject(xlWorkBook);
    releaseObject(xlApp);

    MessageBox.Show("New Worksheet Created!");
}

private void ReadAddPicture_Click(object sender, RoutedEventArgs e) {
    Excel.Application xlApp;
    Excel.Workbook xlWorkBook;
    Excel.Worksheet xlWorkSheet;
    object misValue = System.Reflection.Missing.Value;

    xlApp = new Excel.Application();
    xlWorkBook = xlApp.Workbooks.Add(misValue);
    xlWorkSheet = (Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

    //add some text
    xlWorkSheet.Cells[1, 1] = "Working with Excel!";
    xlWorkSheet.Cells[2, 1] = "Adding picture in Excel File!";

    // This picture must be in this path
    xlWorkSheet.Shapes.AddPicture("d:\\temp\\eTlipse logo 20180708.JPG",
Microsoft.Office.Core.MsoTriState.msoFalse,
Microsoft.Office.Core.MsoTriState.msoCTrue, 50, 50, 300, 45);

    xlWorkBook.SaveAs("d:\\temp\\csharp-Excel.xls",
Excel.XlFileFormat.xlWorkbookNormal, misValue, misValue, misValue, misValue,
Excel.XlSaveAsAccessMode.xlExclusive,
misValue, misValue, misValue, misValue, misValue);
    xlWorkBook.Close(true, misValue, misValue);
}

```

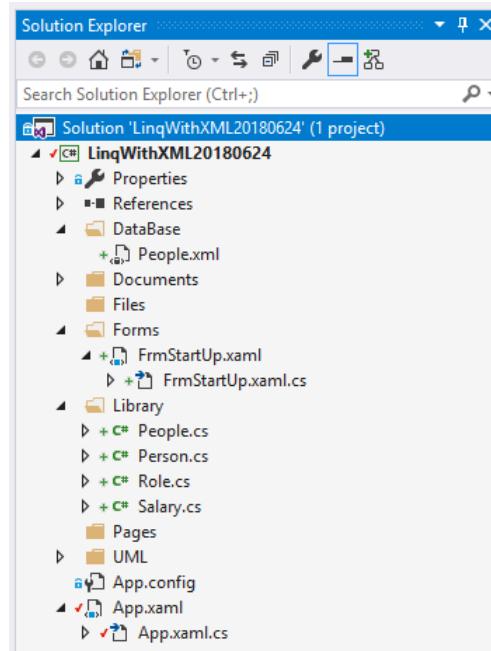
```
    xlApp.Quit();  
  
    releaseObject(xlApp);  
    releaseObject(xlWorkBook);  
    releaseObject(xlWorkSheet);  
  
    MessageBox.Show("File created!");  
}  
  
}  
}
```

Utilizando arquivos XML como Base de Dados

Através deste exemplo entenderemos como interagir com arquivos XML, como fonte de dados. A XML (eXtensible Markup Language) é uma linguagem de marcação que possibilita a criação de documentos contendo dados organizados de forma hierárquica, resultando em uma base de dados bastante versátil. Abriremos o Visual Studio e iniciaremos um novo projeto. Uma vez que este não é o foco neste material, apresentamos o exemplo sem detalhar passo a passo. O projeto ficou organizado como segue.

Exemplo - Interagindo com Arquivo XML

Abriremos o Visual Studio e iniciaremos um novo projeto WPF.



Arquivo base People.xml.

```
<?xml version="1.0" encoding="utf-8"?><people>
  <!--Person section-->
  <person>
    <id>1</id>
    <firstname>Edson</firstname>
    <lastname>Andrade</lastname>
    <idrole>1</idrole>
  </person>
  <person>
    <id>2</id>
    <firstname>Rogerio</firstname>
    <lastname>Lima</lastname>
    <idrole>2</idrole>
  </person>
  <person>
    <id>3</id>
    <firstname>Joel</firstname>
    <lastname>Diniz</lastname>
    <idrole>1</idrole>
  </person>
  <person>
    <id>4</id>
    <firstname>Jade</firstname>
    <lastname>Diniz</lastname>
    <idrole>2</idrole>
  </person>
  <!--Role section-->
  <role>
    <id>1</id>
    <roledescription>Manager</roledescription>
  </role>
  <role>
```

```

<id>2</id>
<roledescription>Developer</roledescription>
</role>
<role>
<id>3</id>
<roledescription>Marketing</roledescription>
</role>
<!--Salary section-->
<salary>
<idperson id="1" year="2001" salaryyear="10000,00" />
<idperson id="1" year="2005" salaryyear="15000,00" />
</salary>
</people>

```

Arquivo xaml da janela principal FrmStartUp.xaml.

```

<Window x:Class="LinqWithXML20180624.Forms.FrmStartUp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:LinqWithXML20180624.Forms"
        xmlns:local_Lib="clr-namespace:LinqWithXML20180624.Library"
        mc:Ignorable="d"
        Title="Linq with XML" Height="450" Width="921">
    <Window.Resources>
        <local_Lib:People x:Key="LocalPeople"/>
    </Window.Resources>
    <StackPanel>
        <!-- Main Menu -->
        <Menu Height="21" Margin="0,0,0,0">
            <MenuItem Header="_File">
                <MenuItem Name="mnuExit" Header="E_xit" Click="Exit"/>
            </MenuItem>
            <MenuItem Header="_Help">
                <MenuItem Header="_About"/>
            </MenuItem>
        </Menu>
        <StackPanel HorizontalAlignment="Center">
            <!-- Data Grid for XML -->
            <TextBlock HorizontalAlignment="Center" Margin="5,9,5,0"
TextWrapping="Wrap" FontSize="12" FontWeight="Bold" Text="Data Table:"/>
            <DataGrid Name="dgrdXML" AutoGenerateColumns="True"
ItemsSource="{StaticResource LocalPeople}" AlternatingRowBackground="LightCyan">
                <!-- Data Grid -->
            </DataGrid>
            <!-- Output -->
            <TextBlock HorizontalAlignment="Center" Margin="5,9,5,0"
TextWrapping="Wrap" FontSize="12" FontWeight="Bold" Text="Output:"/>
            <RichTextBox Name="rtbOutput" HorizontalAlignment="Left" Height="150"
Width="850" Margin="5,9,5,0">
                <FlowDocument>
                    <Paragraph>
                        <Run Text="... output!"/>
                    </Paragraph>
                </FlowDocument>
            </RichTextBox>
        </StackPanel>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right"
Margin="5,9,5,0">

```

```

        <Button x:Name="btnDeleteInXML" Width="105" Margin="3,9,5,5"
Click="BtnDeleteInXML_Click">Delete in XML File</Button>
        <Button x:Name="btnUpDateInXML" Width="107" Margin="3,9,5,5"
Click="BtnUpDateInXML_Click">UpDate in XML File</Button>
        <Button x:Name="btnAddInXML" Width="107" Margin="3,9,5,5"
Click="BtnAddInXML_Click">Add in XML File</Button>
        <Button x:Name="btnAttRetryFromFile" Width="107" Margin="3,9,5,5"
Click="BtnAttRetryFromFile_Click">Retry Att from File</Button>
        <Button x:Name="btnElemRetryFromFile" Width="113" Margin="3,9,5,5"
Click="BtnElemRetryFromFile_Click">Retry Elem from File</Button>
        <Button x:Name="btnDocRetryFromFile" Width="107" Margin="3,9,5,5"
Click="BtnDocRetryFromFile_Click">_Retry Doc from File</Button>
        <Button x:Name="btnApplyETlipseSample" Width="135" Margin="3,9,5,5"
Click="BtnApplyETlipseSample_Click">_Apply Sample ETlipse</Button>
        <Button Name="btnExit" Width="50" Margin="5,9,5,5"
Click="Exit">E_xit</Button>
    </StackPanel>
</StackPanel>
</Window>

```

Arquivo cs da janela principal FrmStartUp.xaml.cs.

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using LinqWithXML20180624.Library;

namespace LinqWithXML20180624.Forms {

    /// <summary>
    /// Lógica interna para FrmStartUp.xaml
    /// </summary>
    public partial class FrmStartUp : Window {

        private void WriteListPeople(List<Person> people) {
            try {

                // Connection with the OutPut Document
                FlowDocument flowDoc = new FlowDocument();
                Paragraph paragraph = new Paragraph();
                paragraph.Inlines.Add(new Bold(new Run("Results:\n\n")));

                // Line to write
                StringBuilder line = new StringBuilder();

```

```

        // Adding all person
        foreach (Person person in people) {
            line.Remove(0, line.Length);
            line.Append(person.ToString());
            line.Append("\n"); paragraph.Inlines.Add(line.ToString());
        }

        // UpDate the OutPut Document
        flowDoc.Blocks.Add(paragraph);
        rtbOutput.Document = flowDoc;

    } catch (Exception ex) {
        MessageBox.Show(ex.ToString());
    }
}

public FrmStartUp() {
    InitializeComponent();
}

private void Exit(object sender, RoutedEventArgs e) {
    this.Close();
}

private void BtnApplyETlipseSample_Click(object sender, RoutedEventArgs e) {

    // Creating People
    List<Person> people = new List<Person> {
        new Person{ID=1, FirstName="Edson", LastName="Andrade", IDRole=1},
        new Person{ID=2, FirstName="Jade", LastName="Diniz", IDRole=2},
        new Person{ID=3, FirstName="Joel", LastName="Diniz", IDRole=1},
        new Person{ID=4, FirstName="Rogerio", LastName="Lima", IDRole=2}
    };

    // Writing Results
    WriteListPeople(people);

    // Feed the DataGrid with people
    dgrdXML.ItemsSource = people;

    // Find people with ID Role = 1
    var query = from q in people
                where q.IDRole == 1
                select q;

    List<Person> selectedPeople;
    selectedPeople = query.ToList<Person>();

    foreach (var p in selectedPeople) {
        MessageBox.Show(p.ToString());
    }
}

private void BtnDocRetryFromFile_Click(object sender, RoutedEventArgs e) {
    try {

        // Connecting with the xml file
        XDocument xml = XDocument.Load(@"..\..\DataBase\People.xml");
    }
}

```

```

// Query a specific data
var query = from p in xml.Elements("people").Elements("person")
            where (int)p.Element("id") == 3
            select p;

string person;

// Interacting with each element
foreach (var record in query) {
    person = string.Format("Person: {0} {1}",
record.Element("firstname").Value, record.Element("lastname").Value);
    MessageBox.Show(person);
}

} catch (Exception ex) {
    MessageBox.Show(ex.ToString());
}
}

private void BtnElemRetryFromFile_Click(object sender, RoutedEventArgs e) {
try {
    XElement xml = XElement.Load(@"..\..\DataBase\People.xml");

    var query = from p in xml.Elements("person")
                where (int)p.Element("id") == 3
                select p;

    string person;

    foreach (var record in query) {
        person = string.Format("Person: {0} {1}",
record.Element("firstname"), record.Element("lastname"));
        MessageBox.Show(person);
    }
} catch (Exception ex) {
    MessageBox.Show(ex.ToString());
}
}

private void BtnAttRetryFromFile_Click(object sender, RoutedEventArgs e) {
try {
    XElement xml = XElement.Load(@"..\..\DataBase\People.xml");

    var query = from s in xml.Elements("salary").Elements("idperson")
                where (int)s.Attribute("year") == 2004
                select s;

    string salary;

    foreach (var record in query) {
        salary = string.Format("Amount: {0}",
(string)record.Attribute("salaryyear"));
        MessageBox.Show(salary);
    }
} catch (Exception ex) {
    MessageBox.Show(ex.ToString());
}
}

private void BtnAddInXML_Click(object sender, RoutedEventArgs e) {
try {

```

```

XmlReader reader = XmlReader.Create(@"..\..\DataBase\People.xml");
XDocument xml = XDocument.Load(reader);
reader.Close();

 XElement idperson = xml.Descendants("idperson").Last();
 idperson.Add(new XElement("idperson",
    new XAttribute("id",1),
    new XAttribute("year",2006),
    new XAttribute("salaryyear","160000,00")));

 // para saída em String
 //StringWriter sw = new StringWriter();
 //XmlWriter w = XmlWriter.Create(sw);
 //xml.Save(w);
 //w.Close();
 //MessageBox.Show(sw.ToString());

 XmlWriter writer = XmlWriter.Create(@"..\..\DataBase\People.xml");
 xml.Save(writer);
 writer.Close();
 MessageBox.Show("Arquivo gravado com sucesso!");

} catch (Exception ex) {
    MessageBox.Show(ex.ToString());
}
}

private void BtnUpDateInXML_Click(object sender, RoutedEventArgs e) {
    try {
        XmlReader reader = XmlReader.Create(@"..\..\DataBase\People.xml");
        XElement xml = XElement.Load(reader);
        reader.Close();

        // Alterando Element
        XElement role = xml.Descendants("role").First();
        MessageBox.Show(role.ToString());

        role.SetElementValue("roledescription", "Manager");
        MessageBox.Show(role.ToString());

        // Alterando Attribute
        role = xml.Descendants("idperson").First();
        MessageBox.Show(role.ToString());

        role.SetAttributeValue("year", "2001");
        MessageBox.Show(role.ToString());

        // Gravando o arquivo
        XmlWriter writer = XmlWriter.Create(@"..\..\DataBase\People.xml");
        xml.Save(writer);
        writer.Close();
        MessageBox.Show("Arquivo gravado com sucesso!");

        // Substituindo elemento completo
        xml.Element("person").ReplaceNodes(new XElement("id", 5),
            new XElement("firstname", "J3di"),
            new XElement("lastname", "Jr"),
            new XElement("idrole", 1));
    } catch (Exception ex) {
        MessageBox.Show(ex.ToString());
    }
}

```

```

        }

    private void BtnDeleteInXML_Click(object sender, RoutedEventArgs e) {
        try {
            // Removendo um elemento
            XmlReader reader = XmlReader.Create(@"..\..\DataBase\People.xml");
            XElement xml = XElement.Load(reader);
            reader.Close();
            xml.Descendants("idperson").First().Remove();
            xml.Elements("role").Remove();
            // Gravando o arquivo
            XmlWriter writer = XmlWriter.Create(@"..\..\DataBase\People.xml");
            xml.Save(writer);
            writer.Close();
            MessageBox.Show("Arquivo gravado com sucesso!");
        } catch (Exception ex) {
            MessageBox.Show(ex.ToString());
        }
    }
}

```

Arquivo da classe People.cs.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinqWithXML20180624.Library {
    class People : List<Person> {
        public People() {
            this.Add(new Person() { ID = 1, FirstName = "Edson", LastName = "Andrade",
IDRole = 1 });
            this.Add(new Person() { ID = 2, FirstName = "Jade", LastName = "Diniz",
IDRole = 2 });
            this.Add(new Person() { ID = 3, FirstName = "Joel", LastName = "Diniz",
IDRole = 1 });
        }
    }
}

```

Arquivo da classe Person.cs.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace LinqWithXML20180624.Library{
    class Person{

        int _id;
        public int ID {
            get { return _id; }
            set { _id = value; }
        }

        int _idRole;
        public int IDRole {
            get { return _idRole; }
            set { _idRole = value; }
        }

        string _lastName;
        public string LastName {
            get { return _lastName; }
            set { _lastName = value; }
        }

        string _firstName;
        public string FirstName {
            get { return _firstName; }
            set { _firstName = value; }
        }

        public override string ToString() {
            return "Person " + ID.ToString() + " - " + FirstName + " " + LastName +
        ".";
        }
    }
}

```

Arquivo da classe Role.cs.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinqWithXML20180624.Library{
    class Role{

        int _id;
        public int ID {
            get { return _id; }
            set { _id = value; }
        }

        string _roleDescription;
        public string RoleDescription {
            get { return _roleDescription; }
            set { _roleDescription = value; }
        }

    }
}

```

}

Arquivo da classe Salary.cs.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinqWithXML20180624.Library{
    class Salary{

        int _idPerson;
        public int IDPerson {
            get { return _idPerson; }
            set { _idPerson = value; }
        }

        int _year;
        public int Year {
            get { return _year; }
            set { _year = value; }
        }

        double _salary;
        public double SalaryYear {
            get { return _salary; }
            set { _salary = value; }
        }
    }
}

```

Arquivo de configuração App.config.

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
    </startup>
</configuration>

```

Arquivo de configuração App.xaml.

```

<Application x:Class="LinqWithXML20180624.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:LinqWithXML20180624"
    Startup="App_Startup">
    <!--StartupUri="MainWindow.xaml"-->

```

```
<Application.Resources>  
    </Application.Resources>  
</Application>
```

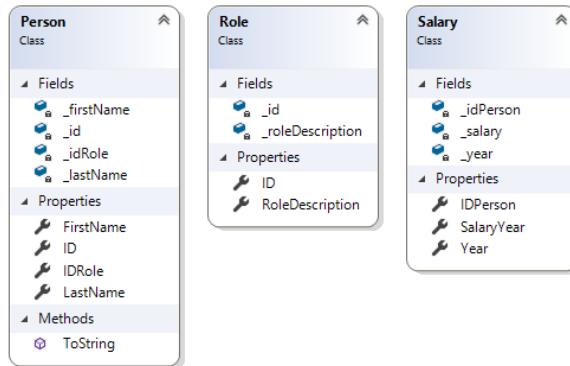
Arquivo de configuração App.xaml.cs.

```
using System;  
using System.Collections.Generic;  
using System.Configuration;  
using System.Data;  
using System.Linq;  
using System.Threading.Tasks;  
using System.Windows;  
  
namespace LinqWithXML20180624 {  
    /// <summary>  
    /// Interação lógica para App.xaml  
    /// </summary>  
    public partial class App : Application {  
        void App_Startup(object sender, StartupEventArgs e) {  
  
            // Open a window  
            Forms.FrmStartUp frmStartUp = new Forms.FrmStartUp();  
            frmStartUp.Show();  
  
            //MainWindow window = new MainWindow();  
            //window.Show();  
        }  
    }  
}
```

UML das classes criadas. UML, Unified Modeling Language, Linguagem de Modelagem Unificada é uma linguagem amplamente utilizada em projetos de software.

UML - Unified Modeling Language

UML das classes criadas. UML, Unified Modeling Language, Linguagem de Modelagem Unificada é uma linguagem amplamente utilizada em projetos de software.

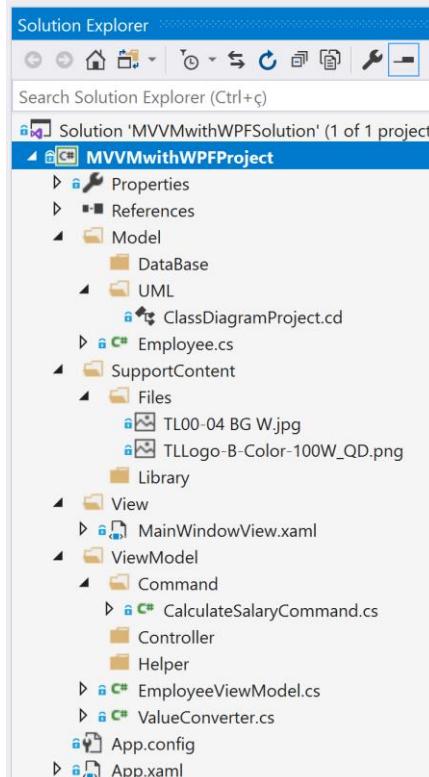


Utilizando padrão MVVM com WPF

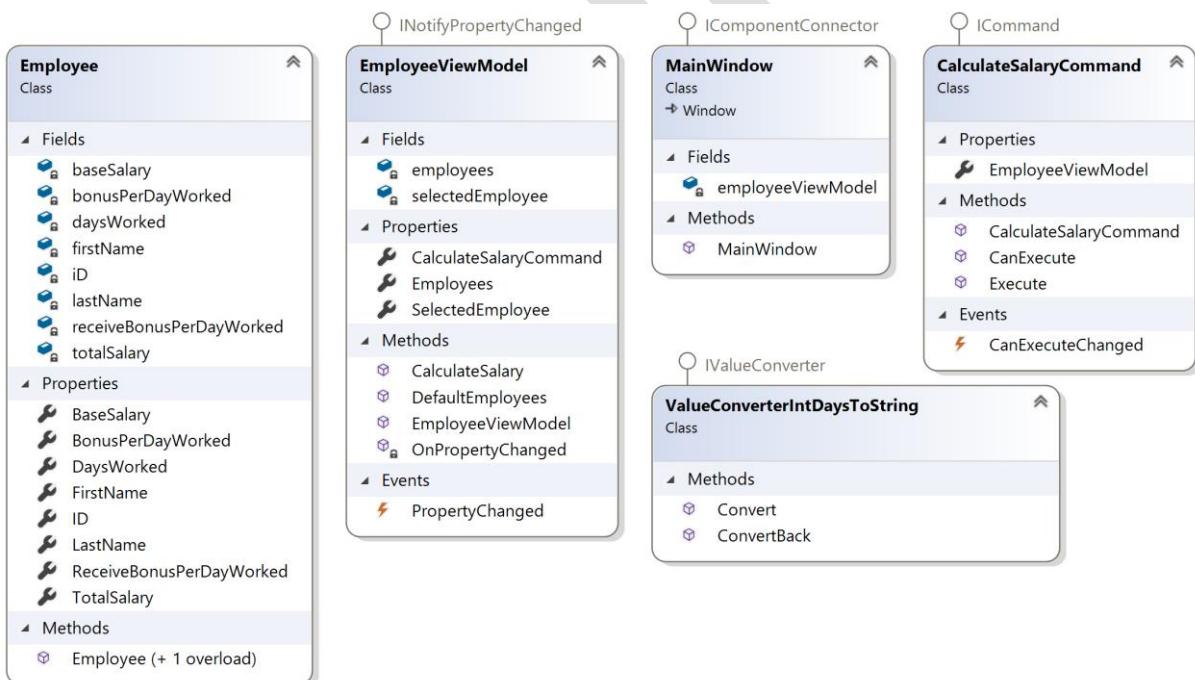
Um importante padrão utilizado originalmente com aplicações WPF é o padrão de projeto Model-View-ViewModel (MVVM). XAML é associada à classes padronizadas que possibilitam a separação da interface do usuário da lógica e modelo da aplicação com recursos como vinculação de dados, o que favorece a implementação deste padrão.

Iremos exemplificar um projeto simples para mostrar o uso do MVVM com WPF. Este exemplo irá mostrar tecnologias importantes a serem utilizadas na programação para separar o modelo da interface.

Criamos pastas para auxiliar na organização do projeto, seguindo a separação de classes e demais recursos de forma a refletir o padrão MVVM. A seguir deixamos uma sugestão de organização de pastas para este padrão.



Um recurso interessante do Visual Studio, para usar UML, é a apresentação do diagrama de classe. A seguir o diagrama de classe que reflete a aplicação.



Nossa classe que representa a lógica da aplicação, alocada na pasta Model, será Employee, e não deve ter ações sobre interface, mas apenas sobre a lógica do negócio. Ela corresponde à abstração de um funcionário, cujo conjunto dará origem a uma lista controlada de funcionários.

Um recurso importante para utilizarmos é o tipo “ObservableCollection”, que define uma coleção que será observada sobre possíveis alterações, o que permite que seus usuários sejam avisados para refletir tais atualizações. No exemplo utilizamos uma classe de empregados que fica sob observação, conforme trecho do código a seguir.

```
private ObservableCollection<Employee> employees;
/// <summary>
```

As classes e recursos de definição de interface devem ficar na pasta View. A pasta ViewModel conterá a camada intermediária que irá comunicar o View, interface, com o Model, modelo do negócio da aplicação. Concentraremos a camada ViewModel em uma classe, definida como EmployeeViewModel.

Um recurso interessante para utilizarmos vem com a implementação da interface “INotifyPropertyChanged”. Este recurso permitirá que a mudança de determinada propriedade seja avisada para que as demais façam ajustes se necessário.

```
5 references
public class EmployeeViewModel : INotifyPropertyChanged {
```

A implementação da interface “INotifyPropertyChanged” requer um evento público “PropertyChangedEventHandler” e um método privado “OnPropertyChanged”. O método passará como string o nome da propriedade que sofreu alteração.

```
public event PropertyChangedEventHandler PropertyChanged;
2 references
private void OnPropertyChanged(string propertyName) {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

As propriedade a serem monitoradas devem chamar o método “OnPropertyChanged” passando o nome da propriedade.

```
public ObservableCollection<Employee> Employees {
    get { return employees; }
    set {
        employees = value;
        OnPropertyChanged("Employees");
    }
}
```

A interface, que ficará alocada na pasta View, será representada pelo XAML e code behind que definem as janelas WPF. O namespace da ViewModel deve ser informado no código XAML e a classe ViewModel deve ser utilizada como recurso no código XAML.

```
xmlns:NamespaceVM="clr-namespace:MVVMwithWPFProject.ViewModel"

    <NamespaceVM:EmployeeViewModel x:Key="employeeViewModel"/>
```

Podemos utilizar um “Grid” para exibir os itens da lista de funcionários. Para que tenhamos o carregamento automático e demais funcionalidades, como atualização, devemos fazer um “Binding” com o ViewModel. Os itens devem receber a ligação com a propriedade correspondente e determinaremos o modo desta ligação, usamos “TwoWay”, o que faz com que tanto a atualização na View seja avisada ao Model assim como o inverso.

```
<Grid x:Name="grdEmployees" DataContext="{Binding employeeViewModel}">

    <ListView ItemsSource="{Binding Employees}"
              SelectedItem="{Binding SelectedEmployee, Mode=TwoWay,
              UpdateSourceTrigger=PropertyChanged}"
              SelectionMode="Single"

    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}"/>
    <GridViewColumn Header="First Name" DisplayMemberBinding="{Binding FirstName}"/>
    <GridViewColumn Header="Last Name" DisplayMemberBinding="{Binding LastName}"/>
    <GridViewColumn Header="Base Salary (R$)" DisplayMemberBinding="{Binding BaseSalary}"/>
```

Este mesmo recurso pode ser utilizado para itens individualizados, não apenas a coleção. Usamos um Employee específico para indicar o que foi destacado, selecionado, da lista. Usamos “SelectedEmployee” para esta funcionalidade. Poderíamos fazer “Binding” diretamente com o “ListView”, mas optamos por organizar o funcionário selecionado no ViewModel para descaracterizar processamento na interface quando possível. O trecho verde esta comentado no XAML.

```
<!-- Optional: -->
<TextBox x:Name="txtName" Text="{Binding ElementName=lstEmployees,Path=SelectedItem.FirstName}" -->
<TextBox x:Name="txtFirstName" Text="{Binding SelectedEmployee.FirstName}" Margin="5,0,5,0"/>
```

Outro recurso importante para evitar code behind na interface do usuário é a implementação da interface “ICommand”, com ela o processamento necessário de ações do usuário pode ser direcionado para a camada ViewModel.

```
3 references
public class CalculateSalaryCommand : ICommand {
```

Para implementação da interface “ICommand” precisamos adicionar o “EventHandler” “CanExecuteChanged”. Precisamos do método público “CanExecute”, este método retorna um “bool” que indicará se está liberada ou não a execução do comando, neste exemplo a aplicação inicia bloqueada até que tenhamos selecionado um Employee. Por fim usamos o método público que executa o comando, caso “CanExecute” seja “true”. Fazemos a chamada do comando implementado no ViewModel para manter nesta classe a responsabilidade do processamento.

```
public event EventHandler CanExecuteChanged {
    add { CommandManager.RequerySuggested += value; }
    remove { CommandManager.RequerySuggested -= value; }
}
```

```

    0 references
    public bool CanExecute(object parameter) {
        string selectedEmployeeName = parameter as string;

        if (string.IsNullOrWhiteSpace(selectedEmployeeName))
            return false;
        return true;
    }

```

```

    0 references
    public void Execute(object parameter) {
        this.EmployeeViewModel.CalculateSalary();
    }

```

Algumas funcionalidades específicas do WPF foram utilizadas para deixar a interface mais interessante, como cores diferentes para “true” ou “false”, mas estas não tem relação com a aplicação do padrão MVVM em princípio.

O código completo ficará como segue, segundo a divisão das classes que foram utilizadas no projeto.

Employee.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MVVMwithWPFProject.Model {
    public class Employee {

        private int iD;
        /// <summary>
        /// Identification number of the employee
        /// </summary>
        public int ID {
            get { return iD; }
            set { iD = value; }
        }

        private string firstName;
        /// <summary>
        /// First name of the employee
        /// </summary>
        public string FirstName {
            get { return firstName; }
            set { firstName = value; }
        }
    }
}

```

```

private string lastName;
/// <summary>
/// Last name of the employee
/// </summary>
public string LastName {
    get { return lastName; }
    set { lastName = value; }
}

private double baseSalary;
/// <summary>
/// Base monthly salary of the employee. Minimal to be received
/// </summary>
public double BaseSalary {
    get { return baseSalary; }
    set { baseSalary = value; }
}

private int daysWorked;
/// <summary>
/// Days worked in current month
/// </summary>
public int DaysWorked {
    get { return daysWorked; }
    set { daysWorked = value; }
}

private bool receiveBonusPerDayWorked;
/// <summary>
/// Receive bonus per day worked?
/// </summary>
public bool ReceiveBonusPerDayWorked {
    get { return receiveBonusPerDayWorked; }
    set { receiveBonusPerDayWorked = value; }
}

private double bonusPerDayWorked;
/// <summary>
/// Bonus per day worked so (Total Salary) = (Days Worked) * (Bonus per Day
Worked) + (Base Salary)
/// </summary>
public double BonusPerDayWorked {
    get { return bonusPerDayWorked; }
    set { bonusPerDayWorked = value; }
}

private double totalSalary;

public double TotalSalary {
    get { return totalSalary; }
    set { totalSalary = value; }
}

public Employee() {

}

public Employee(int iD, string firstName, string lastName, double baseSalary,
int daysWorked, bool receiveBonusPerDayWorked, double bonusPerDayWorked) {
    ID = iD;
    FirstName = firstName;
}

```

```

        LastName = lastName;
        BaseSalary = baseSalary;
        DaysWorked = daysWorked;
        ReceiveBonusPerDayWorked = receiveBonusPerDayWorked;
        BonusPerDayWorked = bonusPerDayWorked;
    }
}
}
}

```

MainWindowView.xaml

```

<Window x:Class="MVVMwithWPFProject.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:MVVMwithWPFProject"
    xmlns:NamespaceVM="clr-namespace:MVVMwithWPFProject.ViewModel"
    mc:Ignorable="d"
    Title="MVVM with WPF Application" Height="450" Width="500"
    Icon="/WpfAppICommandProject;component/SupportContent/Files/TLLogo-B-Color-
100W_QD.png">
    <Window.Background>
        <ImageBrush
            ImageSource="/WpfAppICommandProject;component/SupportContent/Files/TL00-04_BG_W.jpg"/>
    </Window.Background>
    <Window.Resources>
        <NamespaceVM:EmployeeViewModel x:Key="employeeViewModel"/>
        <NamespaceVM:ValueConverterIntDaysToString
x:Key="valueConverterIntDayToString"/>
    </Window.Resources>

    <Grid x:Name="grdEmployees" DataContext="{Binding employeeViewModel}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <ListView ItemsSource="{Binding Employees}"
            SelectedItem="{Binding SelectedEmployee, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
            SelectionMode="Single"
            Name="lstEmployees" Grid.Column="0" Grid.Row="0" Margin="5">
            <ListView.View>
                <GridView x:Name="gvEmployees">
                    <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ID}"/>
                    <GridViewColumn Header="First Name" DisplayMemberBinding="{Binding
FirstName}"/>
                    <GridViewColumn Header="Last Name" DisplayMemberBinding="{Binding
LastName}"/>
                    <GridViewColumn Header="Base Salary (R$)" 
DisplayMemberBinding="{Binding BaseSalary}"/>
                    <GridViewColumn Header="Bonus?">
                        <GridViewColumn.CellTemplate>
                            <DataTemplate>
                                <TextBlock Text="{Binding ReceiveBonusPerDayWorked}">
                                    <TextBlock.Style>

```

```

        <Style TargetType="TextBlock">
            <Style.Triggers>
                <DataTrigger Binding="{Binding
ReceiveBonusPerDayWorked}" Value="False">
                    <Setter Property="Foreground"
Value="Red" />
                <DataTrigger Binding="{Binding
ReceiveBonusPerDayWorked}" Value="True">
                    <Setter Property="Foreground"
Value="Blue" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
</DataTemplate>
</GridViewColumn.CellTemplate>
</GridViewColumn>
</GridView>
</ListView.View>
</ListView>

<StackPanel Grid.Row="0" Grid.Column="1">
    <TextBlock Text="Selected Employee" HorizontalAlignment="Center"
Margin="5" FontWeight="Bold"/>
    <!-- Optional:
    <TextBox x:Name="txtName" Text="{Binding
ElementName=lstEmployees,Path=SelectedItem.FirstName}" Margin="5,0,5,0"/>
    -->
    <TextBox x:Name="txtFirstName" Text="{Binding SelectedEmployee.FirstName}"
Margin="5,0,5,0"/>
    <!-- Optional:
    <TextBlock Text="Last Name" Margin="5"/>
    <TextBox x:Name="txtLastName" Text="{Binding SelectedEmployee.LastName}"
Margin="5,0,5,0"/>
    <!-- Optional:
    <TextBlock Text="Base Salary" Margin="5"/>
    <TextBox x:Name="txtBaseSalary" Text="{Binding
SelectedEmployee.BaseSalary,
    StringFormat={}R$ {0:0.00}}" Margin="5,0,5,0"/>
    <!-- Optional:
    <TextBlock Text="Days Worked" Margin="5"/>
    <TextBox x:Name="txtDaysWorked" Text="{Binding
SelectedEmployee.DaysWorked,
    Converter={StaticResource valueConverterIntDayToString}}"
Margin="5,0,5,0"/>
    <!-- Optional:
    <TextBlock Text="Bonus per Day Worked" Margin="5"/>
    <TextBox x:Name="txtBonusPerDayWorked" Text="{Binding
SelectedEmployee.BonusPerDayWorked,
    StringFormat={}R$ {0:0.00}}" Margin="5,0,5,0"/>
    <!-- Optional:
    <Separator Margin="5,18,5,15"/>
    <!-- Optional:
    <Button x:Name="btnTotalSalary" Margin="5" Command="{Binding
CalculateSalaryCommand}"
        CommandParameter="{Binding SelectedEmployee.FirstName}>
        _Compute Total Salary
    </Button>

```

```

<!-->
<TextBlock Text="Total Salary" Margin="5"/>
<TextBox x:Name="txtTotalSalary" Text="{Binding
SelectedEmployee.TotalSalary, Mode=TwoWay,
StringFormat={}>R$ {0:0.00}}" Margin="5,0,5,0"/>
</StackPanel>

</Grid>
</Window>

```

MainWindowView.xaml.cs

```

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using MVVMwithWPFProject.Model;
using MVVMwithWPFProject.ViewModel;

namespace MVVMwithWPFProject {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window {
        private EmployeeViewModel employeeViewModel;
        public MainWindow() {

            InitializeComponent();

            employeeViewModel = new EmployeeViewModel();

            // Adding itens just to test
            employeeViewModel.DefaultEmployees();
            grdEmployees.DataContext = employeeViewModel;
        }
    }
}

```

CalculateSalaryCommand.cs

```

using MVVMwithWPFProject.ViewModel;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

```

```

using System.Windows.Markup;

namespace WpfAppICommandProject.ViewModel.Command {
    public class CalculateSalaryCommand : ICommand {

        public EmployeeViewModel EmployeeViewModel { get; set; }

        public event EventHandler CanExecuteChanged {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public CalculateSalaryCommand(EmployeeViewModel employeeViewModel) {
            EmployeeViewModel = employeeViewModel;
        }

        public bool CanExecute(object parameter) {
            string selectedEmployeeName = parameter as string;

            if (string.IsNullOrWhiteSpace(selectedEmployeeName))
                return false;
            return true;
        }

        public void Execute(object parameter) {
            this.EmployeeViewModel.CalculateSalary();
        }
    }
}

```

EmployeeViewModel.cs

```

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using MVVMwithWPFProject.Model;
using WpfAppICommandProject.ViewModel.Command;

namespace MVVMwithWPFProject.ViewModel {
    public class EmployeeViewModel : INotifyPropertyChanged {

        private ObservableCollection<Employee> employees;
        /// <summary>
        /// Collection of employees
        /// </summary>
        public ObservableCollection<Employee> Employees {
            get { return employees; }
            set {
                employees = value;
                OnPropertyChanged("Employees");
            }
        }

        private Employee selectedEmployee;
        /// <summary>

```

```

/// Selected employee from the collection of employees
/// </summary>
public Employee SelectedEmployee {
    get { return selectedEmployee; }
    set {
        selectedEmployee = value;
        OnPropertyChanged("SelectedEmployee");
    }
}

public EmployeeViewModel() {
    //SelectedEmployee = new Employee(1, "Joel", "Diniz", 5000, 20, true,
100f);
    CalculateSalaryCommand = new CalculateSalaryCommand(this);
}

/// <summary>
/// Default contractor to include employees to test
/// </summary>
public void DefaultEmployees() {
    employees = new ObservableCollection<Employee>(){
        new Employee(1, "Joel", "Diniz", 5000, 20, true, 100f),
        new Employee(2, "Edson", "Andrade", 10000, 10, true, 100f),
        new Employee(3, "Jade", "Diniz", 1000, 0, false, 0f)
    };
    this.Employees = employees;
}

public CalculateSalaryCommand CalculateSalaryCommand { get; set; }

public void CalculateSalary() {
    if (SelectedEmployee != null) {

        // (Total Salary) = (Days Worked) * (Bonus per Day Worked) + (Base
Salary)
        SelectedEmployee.TotalSalary = (SelectedEmployee.DaysWorked *
SelectedEmployee.BonusPerDayWorked)+SelectedEmployee.BaseSalary;
        this.SelectedEmployee = SelectedEmployee;

        // Teste if this employee receives bonus
        if (!SelectedEmployee.ReceiveBonusPerDayWorked) {
            MessageBox.Show("This employee does not receive bonus!");
        }
    } else {
        MessageBox.Show("No employee selected!");
    }
}

public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

ValueConverter.cs

```

using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Data;

namespace MVVMwithWPFProject.ViewModel {
    /// <summary>
    /// Converts the days, that are int, to string format
    /// </summary>
    public class ValueConverterIntDaysToString : IValueConverter {
        public object Convert(object value, Type targetType, object parameter,
CultureInfo culture) {
            int inputAsInt = int.Parse(value.ToString());
            if (inputAsInt > 0) {
                return inputAsInt.ToString() + " days";
            } else {
                return inputAsInt.ToString() + " day";
            }
        }

        public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture) {
            var input = value as string;
            int output;
            if (int.TryParse(input, out output))
                return output;
            else
                return DependencyProperty.UnsetValue;
        }
    }
}

```

App.xaml

```

<Application x:Class="MVVMwithWPFProject.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:MVVMwithWPFProject"
    StartupUri="View/MainWindowView.xaml">
    <Application.Resources>
        </Application.Resources>
    </Application>

```

Revit API

Desenvolvimento de plug-ins com a API, Application Programming Interface, do Revit, utilizando a linguagem C#, do .Net, no Visual Studio.

Utilizaremos, como recurso, o “Revit 2019” e o “Visual Studio 2017” para desenvolver nossos exemplos. Estas duas plataformas podem ser adquiridas na versão estudante, para uso educacional.

O Visual Studio deve ser otimizado para trabalhar com C#, linguagem computacional, e utilizar o Inglês como linguagem, para melhor acompanhamento dos exemplos.

Nem todos os passos definidos são indispensáveis para o funcionamento de todos os exemplos, mas sua definição é importante para uma boa prática de programação ou sendo mesmo indispensáveis para exemplos específicos.

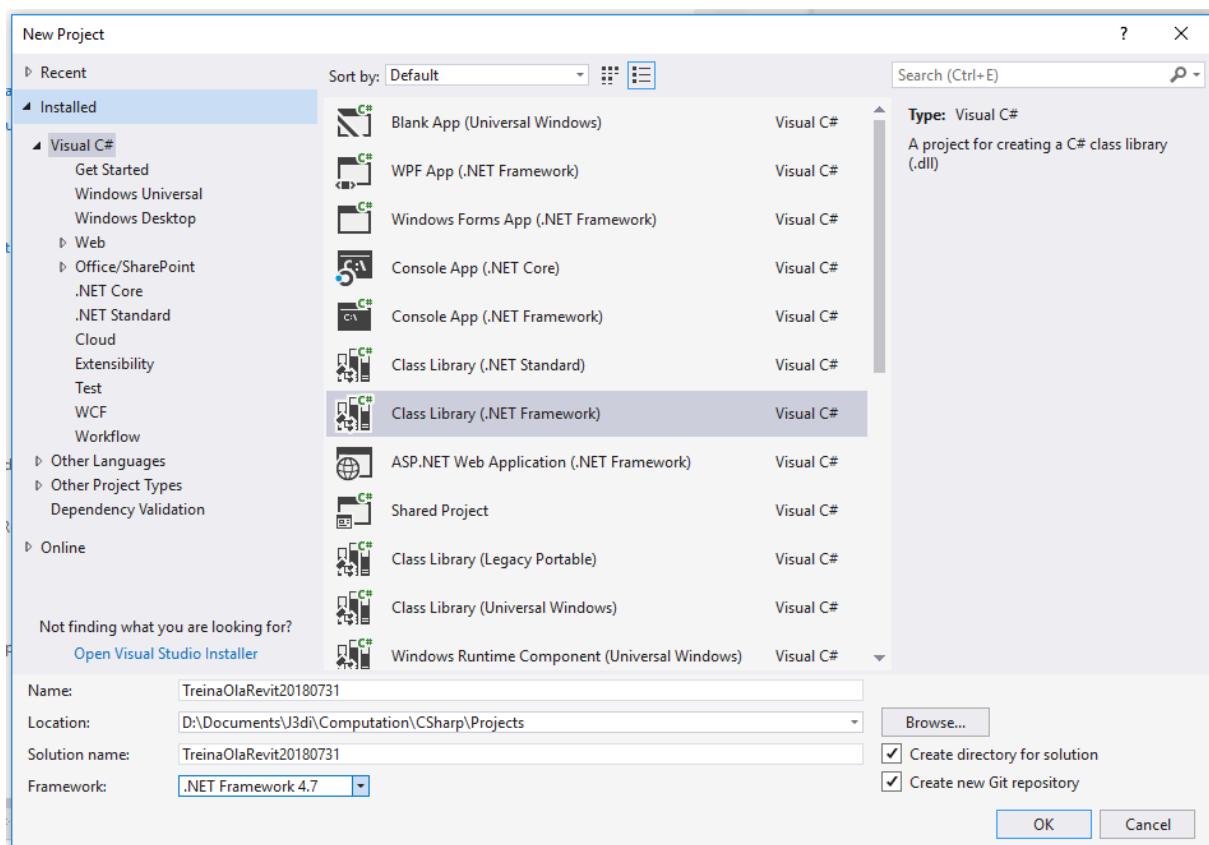
Usaremos exemplos práticos para entender os conceitos necessários para o uso da API do Revit, com a linguagem C#, na plataforma Visual Studio.

É possível encontrar ajuda específica no SDK, software development kit, da API, disponível com a instalação do pacote do Revit, sendo o instalador do SDK, que fica disponível apenas nos arquivos de instalação, encontrado em “C:\Autodesk\Revit_2019_G1_Win_64bit_dlm\Utilities\SDK\RevitSDK.exe”. O SDK traz ajuda, bem como exemplos e arquivos de suporte para o desenvolvimento utilizando a API do Revit.

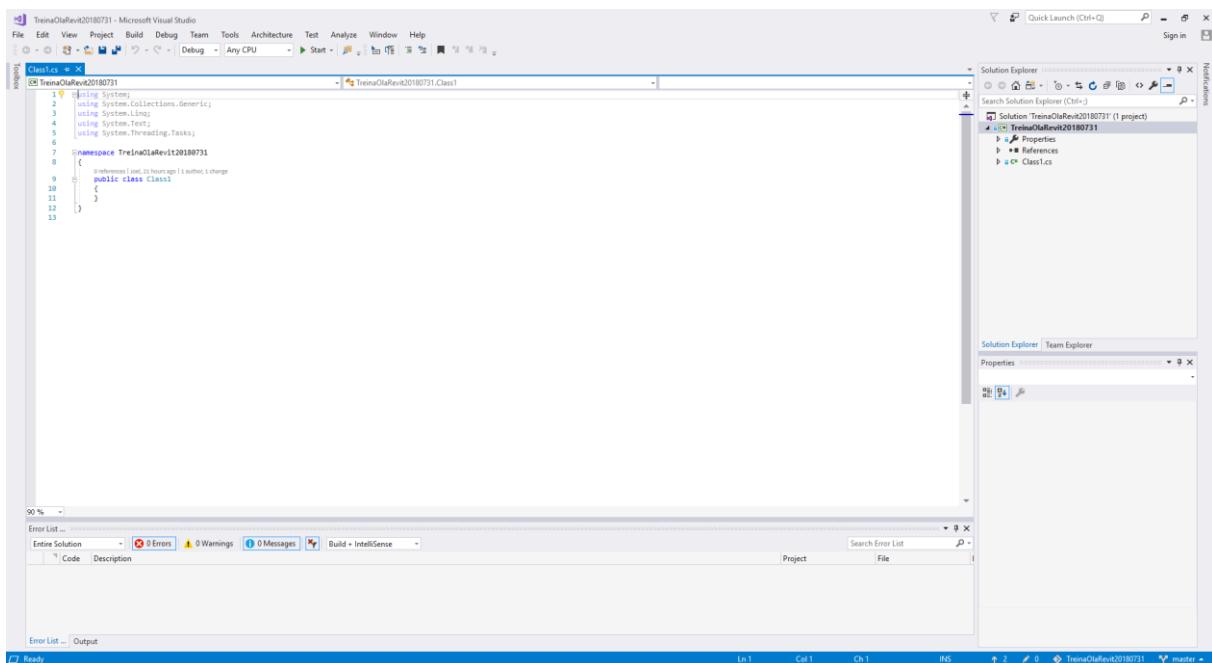
Exemplo - Projeto “Olá Revit!”

Existem vários tipos de projetos que podem ser criados com o Visual Studio. Utilizaremos Class Library para criar um plug-in para o Revit.

- Menu File
- New
- Project



Uma janela para as definições do projeto possibilita escolher os padrões a serem utilizados. Optaremos por “Visual C#” para linguagem, “Class Library” para gerar uma dll, dynamic link library. O nome do projeto será “TreinaOlaRevit20180731”. Voltamos para a janela principal, e o projeto criado é exibido.



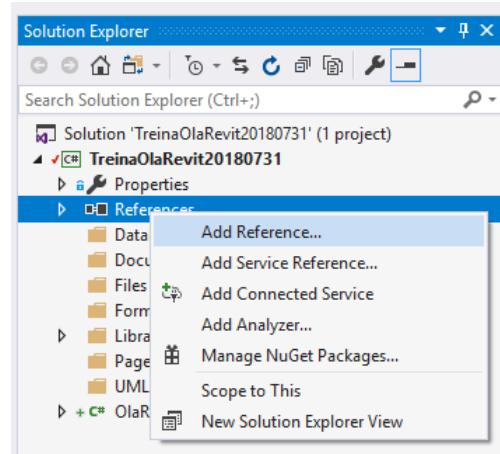
O arquivo “Class1.cs” contém nosso código C#. Iremos renomear esse arquivo para “OlaRevit.cs”.

Para isso, deveremos clicar com o botão direito sobre o arquivo e escolher a opção “Renomear”. O Visual Studio deve perguntar se desejamos atualizar as referências a este nome, e devemos escolher “Sim”.

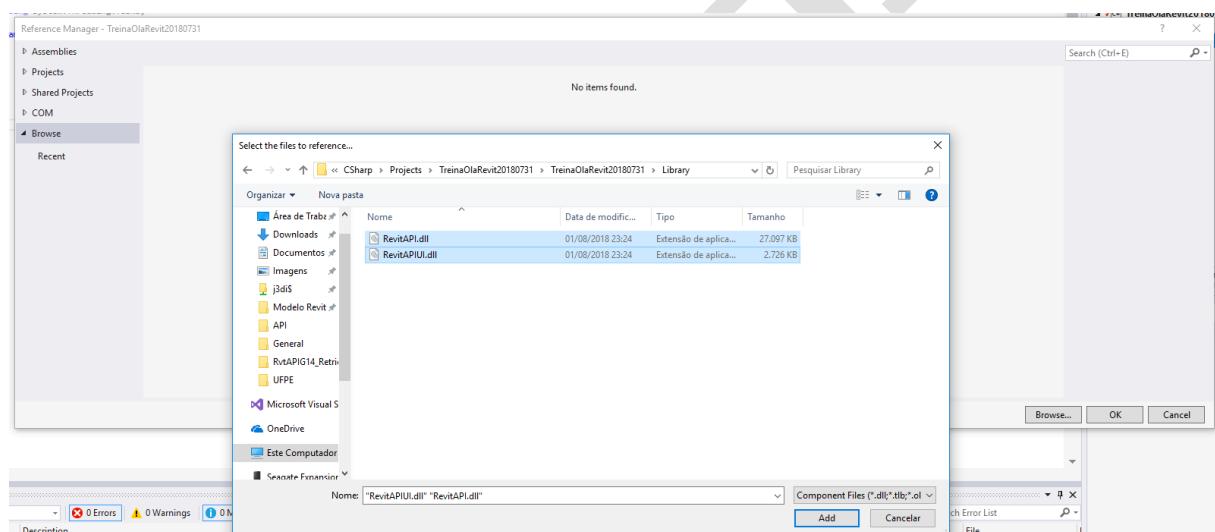
Não se deve reinventar a roda na programação. Faremos uso de bibliotecas de códigos, que contém o código necessário para reuso e montagem da nossa lógica. A API do Revit está contida em dll, dynamic link library, que são bibliotecas com a descrição do código necessário para interagir com o Revit. Precisamos adicionar referência às bibliotecas que usaremos.

Primeiramente adicionaremos as duas dll necessárias para usar a API do Revit, sendo “RevitAPI.dll” e “RevitAPIUI.dll”. Eles estão no diretório onde nosso Revit está instalado, possivelmente “C:\Program Files\Autodesk\Revit 2019”.

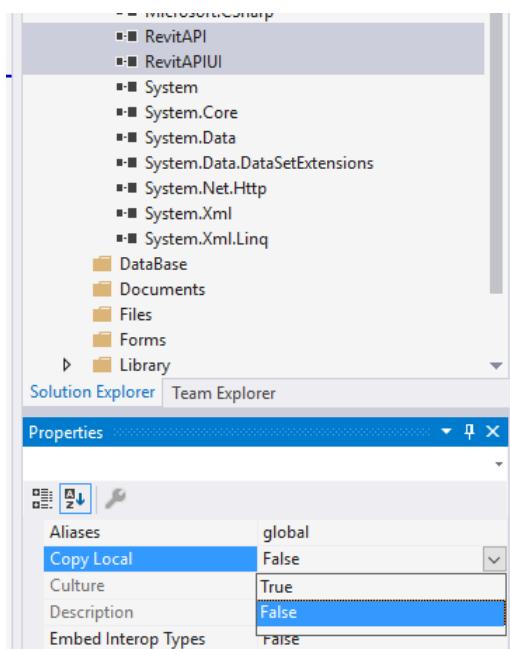
Clicamos com o botão direito do mouse em References, da janela Solution Explorer. Selecioneamos “Add Reference...”



Use a opção de “Browser” para navegar até o local onde estão as dlls e adicione. Uma boa opção pode ser antes copiá-las para a pasta “Library” do projeto, isto porque mudanças de instalação do Revit podem modificar ou perder o local original destas bibliotecas da API.



Altere no Solution Explorer o parâmetro “Copy Local” para falso. Isto vai evitar que uma cópia da biblioteca seja copiada para a pasta sempre que o programa for copiado.



Agora precisamos fazer algumas modificações em nosso código C#, arquivo “*OlaRevit.cs”, de tal forma que possa atender às exigências da API.

Adicionamos as bibliotecas que o código fará uso:

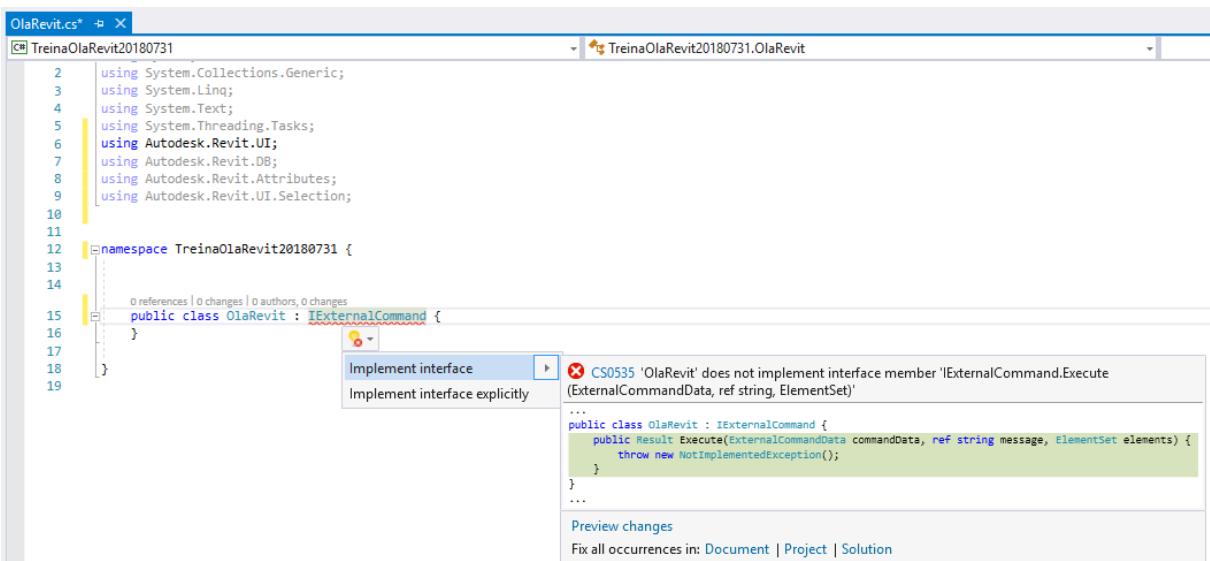
```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
```

Adicionamos a directive que instrui sobre a transação. Atentar que nas atualizações a opção Automática não é mais suportada:

```
[Transaction(TransactionMode.Manual)]
```

Precisamos implementar a interface `IExternalCommand`, e neste caso faremos uso de uma funcionalidade bem interessante do Visual Studio. Adicione o código “`: IExternalCommand`” logo após à nossa definição da classe. O Visual Studio irá nos possibilitar definir a implementação desta interface,

bastando usar a opção que irá aparecer sob a definição, um ícone de uma lâmpada com a seta que selecionaremos, e que aparece quando deixamos a seleção sobre o código recém adicionado, conforme imagem a seguir.



Um método, “Execute”, será adicionado, e, uma vez que ele retorna “Result”, precisamos dar esta saída ao nosso código. Podemos comentar o código que lança a exceção de não implementação do código com “//”. Usaremos a saída dentro de uma estrutura “try … catch”, para tratar exceções.

Este nosso código apenas exibirá uma mensagem no Revit com uso de janela de texto normal, o que pode ser feito com o código “`TaskDialog.Show("Revit", "Olá Revit!");`”.

Nosso código completo ficou:

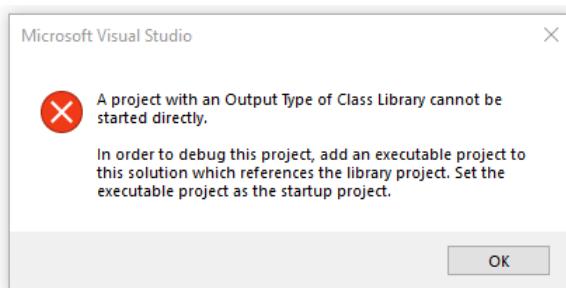
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace TreinaOlaRevit20180731 { // NameSpace.
```

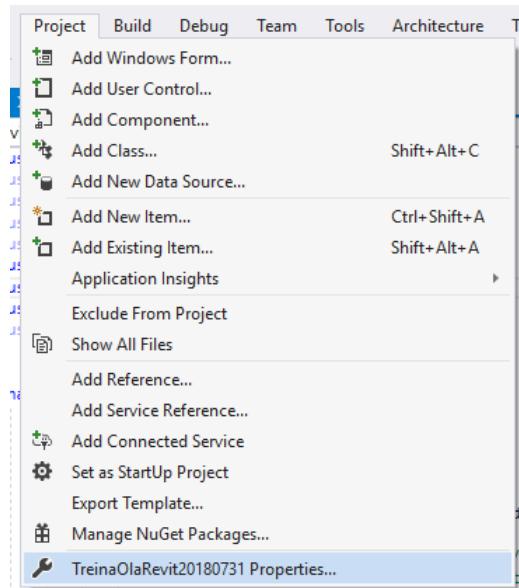
```
[Transaction(TransactionMode.Manual)]
public class OlaRevit : IExternalCommand { // Classe que irá dar origem à dll para
uso no Revit.
    public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) { // Método de entrada que implementa a interface necessária da
API.
        try {
            TaskDialog.Show("Revit", "Olá Revit!"); // Caixa de diálogo que será
executada se tudo der certo.
            return Result.Succeeded; // Retorno que tudo ocorreu bem.
        } catch (Exception ex) { // Tratamento de exceção.
            message = ex.Message;
            TaskDialog.Show("Error!", message);
            return Result.Failed; // Retorno de erro.
            //throw; // Código a ser comentado.
        }
        //throw new NotImplementedException(); // Código a ser comentado.
    }
}
```

Devemos então compilar o arquivo . E observar se não obtivemos erro na barra de status . Uma tela de advertência pode ser apresentada, visto que não estamos criando um programa, e não atribuímos a um aplicativo, o que pode ser configurado para usar o Revit, mas ainda não o faremos aqui.

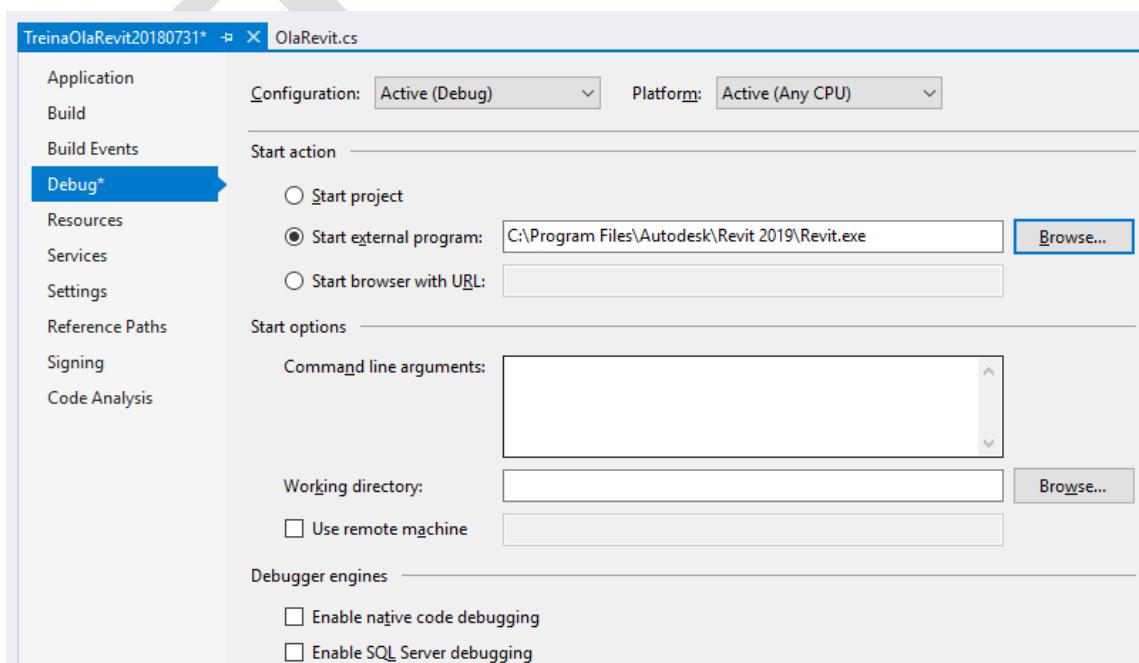
Quando executamos o projeto desta forma, teremos a advertência de que a saída não pode ser executada diretamente, visto ser uma dll, e que precisaríamos de um programa externo para isso. Para testarmos imediatamente a biblioteca, podemos então configurar o Revit como o programa para testar a saída.



O programa de saída deve ser configurado a partir do menu “Project”, submenu “Properties...”, que nos remeterá à tela de configuração.



Devemos então a aba “Debug”, e então optar por “Start external program:”. O botão “Browser” nos permite selecionar o programa que será executado quando compilarmos o projeto, no caso o Revit. Navegue até a pasta onde o Revit foi instalado e selecione o executável “Revit.exe”. Um provável caminho será “C:\Program Files\Autodesk\Revit 2019”.



Sempre que compilarmos o projeto o Revit será inicializado, caso não tenhamos algum erro a depurar. A biblioteca, o arquivo dll, será criado na pasta “\bin\Debug\” do nosso projeto. Esta é a dll que consiste em nosso plug-in e que será apontada para que o Revit possa fazer uso do nosso código, algo que é feito com um manifesto.

Elaboração de Manifesto que Habilita o Plugin

O Revit, ao iniciar, checará se existe um arquivo, que denominamos manifesto, na sua pasta padrão de plug-ins. Caso exista um arquivo nesta pasta, o Revit irá interpretar. Esta é a forma de permitirmos o uso de plug-ins no Revit. Caso tudo ocorra bem com a interpretação, seremos avisados do seu carregamento, e teremos que optar entre não carregar a dll, até porque está é uma possibilidade de entrada para vírus, carregar apenas uma vez, ou sempre carregar. Quando optamos por carregar sempre, o Revit não nos dará avisos de carregamento deste plug-ins nas próximas inicializações. Um exemplo de manifesto está na área de download da eTlipse, site <https://www.etlipse.com/>, que se refere ao uso do AddInManager, um aplicativo que possibilita carregar diretamente plug-ins do formato .dll.

Criaremos o arquivo “TreinaOlaRevit20180731.addin”, que conterá as informações necessárias para interpretação na inicialização do Revit. Este arquivo precisará estar na pasta de plug-ins do Revit, em geral, “C:\Users\[XXXX]\AppData\Roaming\Autodesk\Revit\Addins\2020\”, onde “[XXXX]” é o nome do usuário. A pasta AppData pode ser encontrada mais facilmente pelo atalho “%AppData%”.

Nosso arquivo ficará conforme modelo abaixo. Os parênteses usamos aqui apenas para comentar, mas obviamente não devem estar contidos em nosso manifesto.

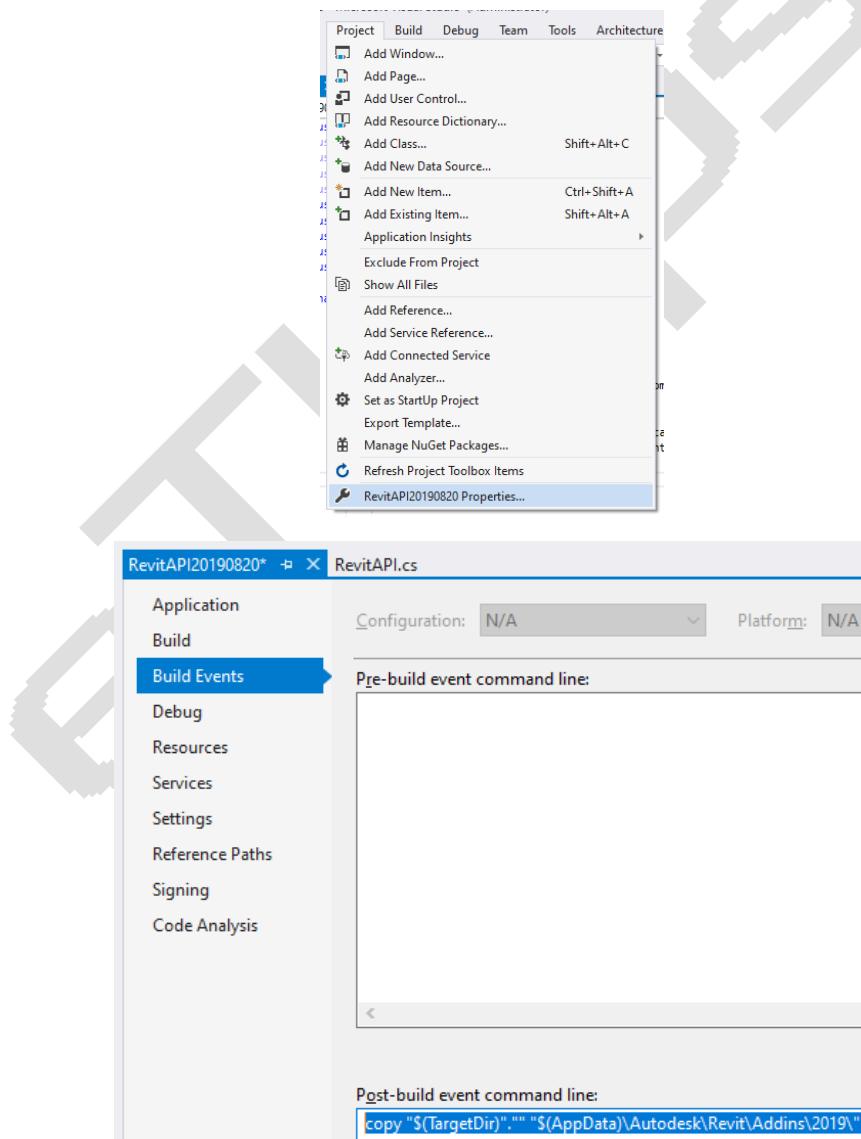
```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Name>SampleApplication</Name>
    <Assembly>C:\Temp\RevitPlugins\ TreinaOlaRevit20180731.dll</Assembly> (Arquivo e
Local onde copiaremos nossa dll, resultado do arquivo copiado)
    <AddInId>BC05297E-11B0-4CCC-A0F4-35AFC0DD12E7</AddInId> (Código único de Id do
Plugin, que pode ser gerado no Visual Studio)
    <FullClassName> TreinaOlaRevit20180731.OlaRevit</FullClassName> (NameSpace e nome
da classe que criamos)
    <VendorId>ETlipse</VendorId>
    <VendorDescription>Edson Andrade, Rogerio Lima, Joel Diniz</VendorDescription>
  </AddIn>
</RevitAddIns>
```

O Id do plug-in deve ser único, e pode ser gerado no próprio Visual Studio, em Tools > Create GUID > 5. Este é um número gerado com algoritmo especial para ser improvável a coincidência.

Uma opção muito interessante, para etapa de testes, pode ser usar o plug-in AddInManager, que vinha com versões anteriores do SDK da API do Revit. Suficiente copiar a dll e o addin na pasta de plugins. Uma versão está disponível na área de download da eTLipse.

É possível automatizar a cópia do Manifesto para pasta adequada do Revit, que entende os plugins a serem executados. Devemos acessar, no Visual Studio, o menu **Project**, submenu **Properties**. Na janela de propriedades do projeto, na aba Built Events, caixa Post-build, colocamos a directiva abaixo, que dependerá do Revit em uso, no exemplo o 2019.

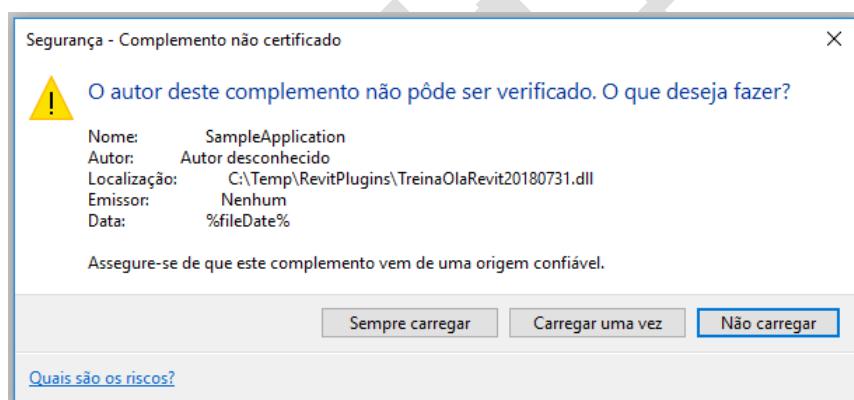
```
copy "$(TargetDir)"" "" $(AppData)\Autodesk\Revit\Addins\2019\"
```



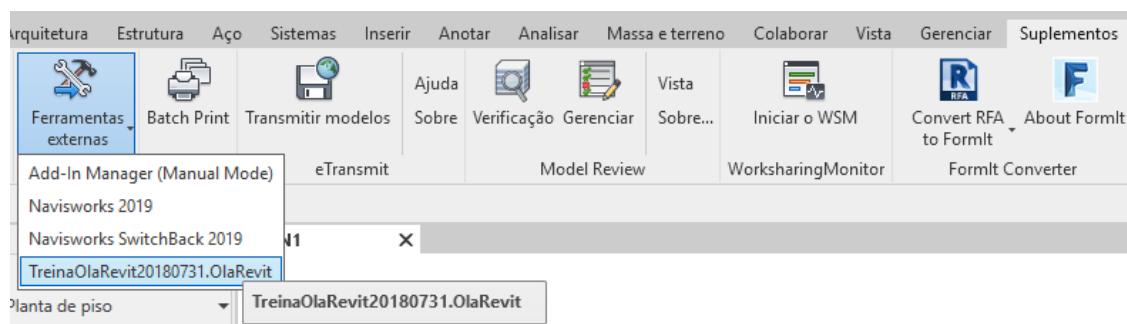
Conforme citado, o Revit irá adicionar como plugin os arquivos processados na pasta citada de plugins. Estes arquivos basicamente consistem no addin, que precisa estar corretamente configurado apontando para a *dll*, e a *dll* em si que irá conter o código do plugin. Caso opte por sempre abrir o plugin, no seu primeiro acesso, estes arquivos irão permanecer na pasta padrão, e não será mais lançada a pergunta sobre seu processamento, o que passará a ocorrer automaticamente. Na fase de teste é portanto interessante optar pelo carregamento temporário. Caso precise desinstalar seu plugin que optou por sempre carregar basta deletar ou mover os arquivos da referida pasta.

Executando o plug-in

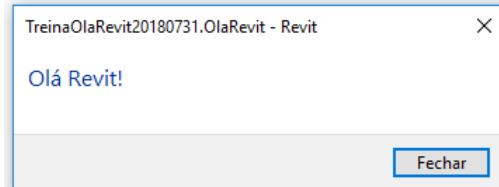
Inicializamos o Revit, e obteremos o aviso que descrevemos. Para teste pode ser interessante ainda não optar por aceitar sempre executar o plug-in.



O plug-in que criamos estará disponível em Suplementos > Ferramentas externas. Este será o padrão, mas é possível criar novos itens e menus.



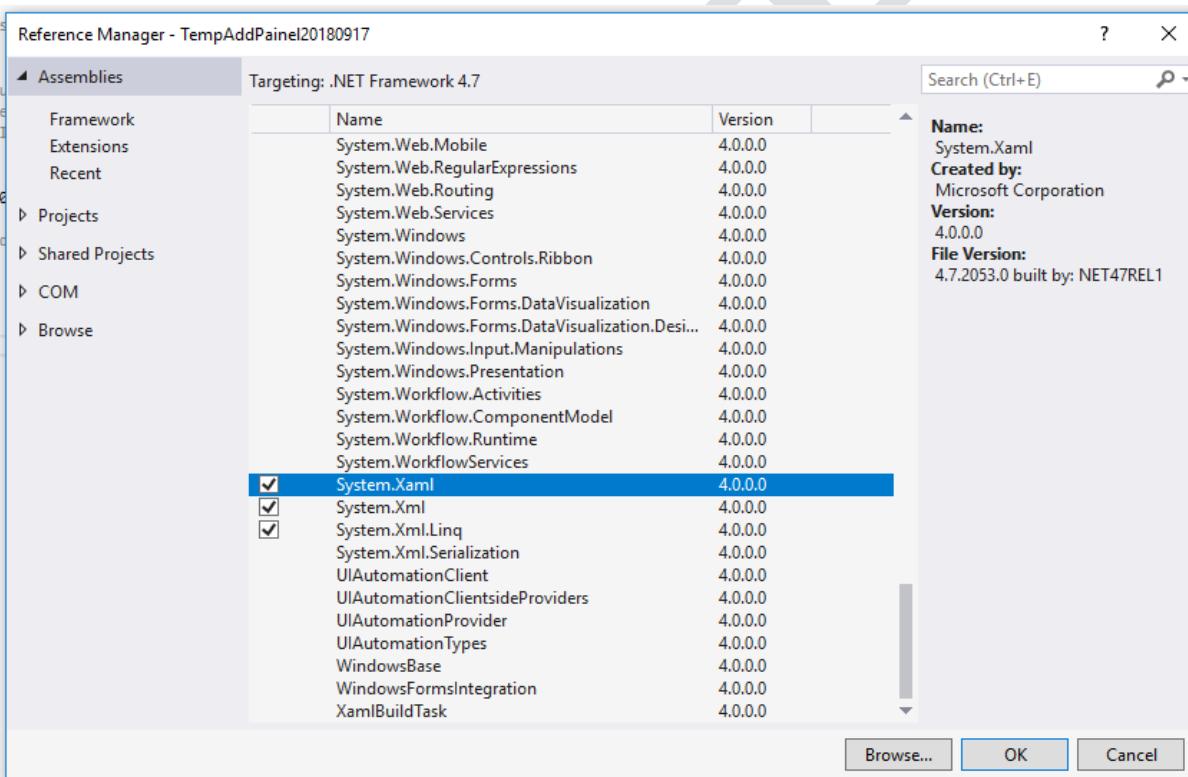
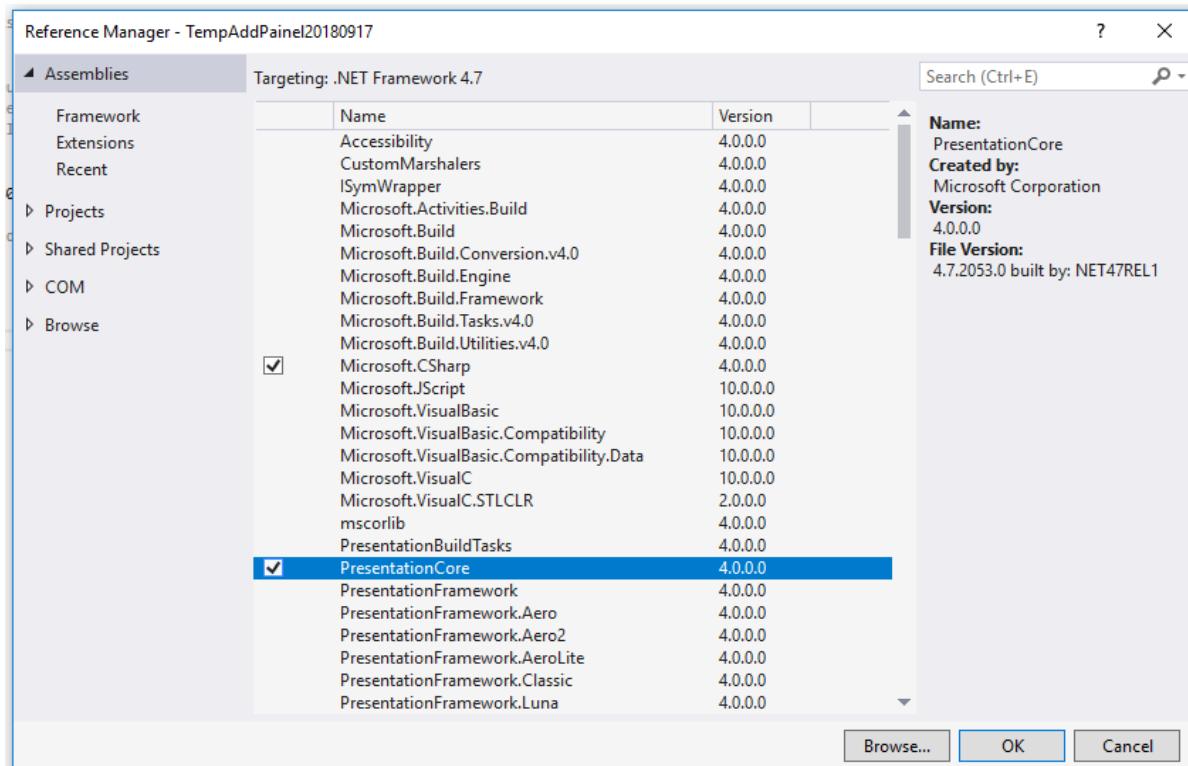
Conforme pretendido, a execução, com o clique no item, nos remeterá à tela de diálogo com texto que programamos.



Personalizar Ribbon Painel

Para criação de plugin é possível personalizar o painel Ribbon do Revit e configurar ícones que direcionem o usuário aos plug-ins personalizados.

Criamos um projeto no Visual Studio, de nome “AddPanel”, conforme no primeiro exemplo. Desta vez precisaremos adicionar mais referências. Devemos adicionar referência a “PresentationCore”. O mesmo deve ser feito para adicionar “System.Xaml”.



Renomeamos “Class1.cs” para “CsAddPanel.cs”, a partir de Solution Explorer, botão direito do mouse sobre o arquivo, em seguida opção renomear. Abrimos o arquivo “CsAddPanel.cs” para editá-lo conforme abaixo. Diferente da primeira aplicação, que era baseada em Command, está é baseada em

Application, e contém dois métodos abstratos, OnStartup() e OnShutdown(). É preciso definir uma imagem de ícone e usar seu caminho no código, bem como usar o caminho da dll, conforme código a seguir. O botão criado deve apontar para a dll que se pretende executar quando este for acionado no Revit. O arquivo de manifesto deve ser criado de forma semelhante ao feito anteriormente, desta vez com tipo “Application”.

O código completo fica:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
using System.Windows.Media.Imaging;

namespace RevitAPIGuide2014_AddPanel_20180708{

    [Transaction(TransactionMode.Manual)]
    public class CsAddPanel : Autodesk.Revit.UI.IExternalApplication {

        Result IExternalApplication.OnShutdown(UIControlledApplication application) {
            return Result.Succeeded;
        }

        Result IExternalApplication.OnStartup(UIControlledApplication application) {
            // Add new ribbon panel
            RibbonPanel ribbonPanel = application.CreateRibbonPanel("NewRibbonPanel");

            // Create a push button in the ribbon panel "NewRibbonPanel"
            // the add-in application "Helloworld" will be triggered when button is
pushed

            PushButton pushButton = ribbonPanel.AddItem(new
PushButtonData("Helloworld", "Helloworld",
@"C:\Temp\RevitPlugins\RevitAPIGuid2014_20180702.dll",
"RevitAPIGuid2014_20180702.HelloWorld")) as PushButton;

            // Set the large image shown on button
            Uri uriImage = new Uri(@"D:\Temp\Temp_32x32.png");
            BitmapImage largeImage = new BitmapImage(uriImage);
            pushButton.LargeImage = largeImage;

            return Result.Succeeded;
        }
    }
}

```

O arquivo de manifesto será:

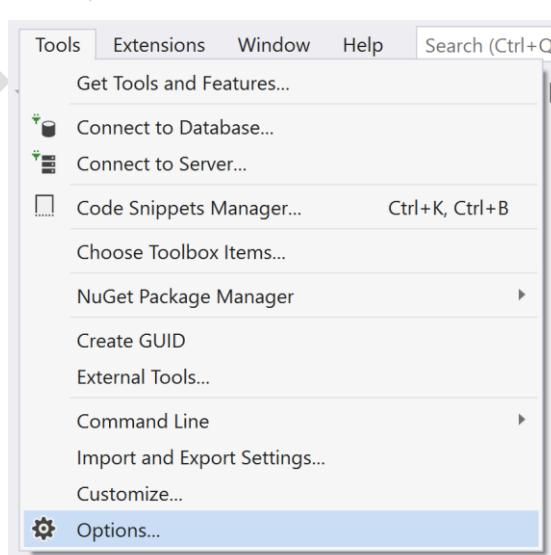
```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Application">
    <Name>SampleApplication</Name>
    <Assembly>C:\Temp\RevitPlugins\RevitAPIGuide2014_AddPanel_20180708.dll</Assembly>
    <AddInId>BC05297E-11B0-4CCC-A0F4-35AFC0DD12E7</AddInId>
    <FullClassName>RevitAPIGuide2014_AddPanel_20180708.CsAddPanel</FullClassName>
    <VendorId>ETLipse</VendorId>
    <VendorDescription>Edson Andrade, Rogerio Lima, Joel Diniz</VendorDescription>
  </AddIn>
</RevitAddIns>
```

Revit API com WPF - Template eTLipse.

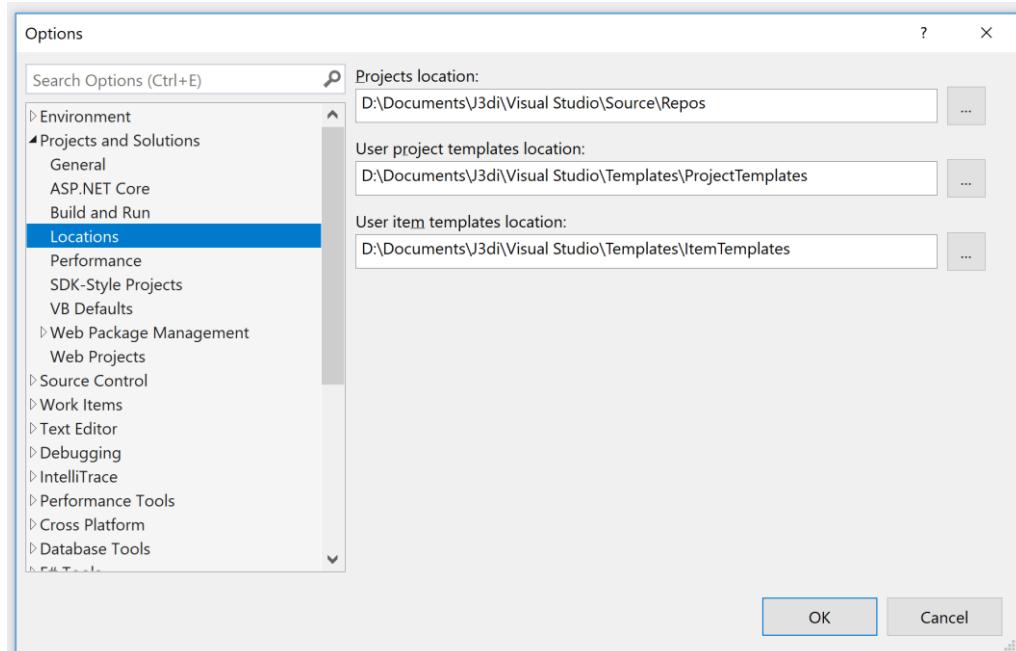
Algumas configurações são necessárias para fazer uso de interface WPF com a API do Revit, algumas bem específicas e não tão usuais. A eTLipse disponibiliza um template específico para esta necessidade, facilitando o uso deste importante recurso da plataforma .Net.

O template da eTLipse pode ser baixado na página da eTLipse para ser utilizado no Visual Studio e facilitar o uso e configuração de aplicação WPF utilizando a API do Revit. O arquivo do template consiste em um arquivo comprimido em formato .zip que deve ser colocado no diretório apropriado. Importante verificar o diretório padrão utilizado pela versão em uso do Visual Studio.

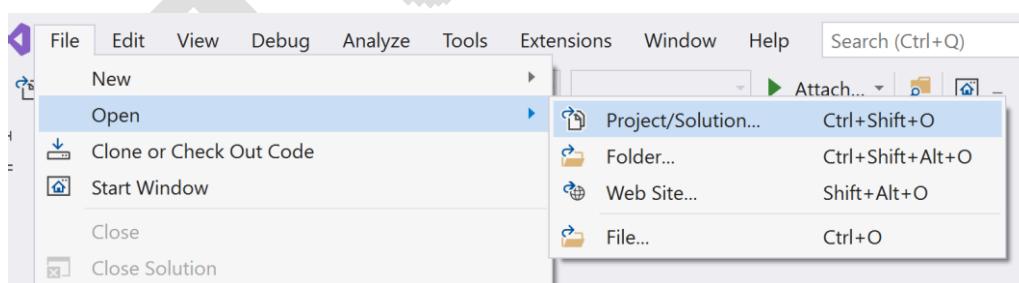
A configuração do caminho utilizado para carregar os templates está registrada no menu de configuração. Utilize a opção “Options...” do menu “Tools”.



Na janela de opções devemos escolher o item “Projects e Solutions”, e, como subtópico desta opção, teremos o item “Locations”. Será exibida as opções de caminho em “Projects Location” nesta mesma janela, onde devemos observar o item “User Project templates location”.



O arquivo de configuração deverá ser colocado dentro desta página para ser carregado como template pelo Visual Studio. Feita a cópia para a pasta correta, teremos a possibilidade de criar um novo projeto baseado no template escolhido. Um novo projeto podemos iniciar a partir do item “Projeto” da submenu “New” do menu “File”.



Podemos observar que o Visual Stúdio irá apresentar o template da eTLipse como uma das opções para novo projeto, conforme figura a seguir. Precisamos selecionar a opção exibida “TL Revit 2020 WPF Addin” e clicar em “Next”.

Create a new project

Recent project templates



All languages ▾ All platforms ▾ All project types ▾

- TL Revit 2020 WPF Addin
 eTlipse Revit 2020 Add-In WPF project for an application with multiple commands in a modeless dialog.
C# WPF Revit Add-In
- WPF Library (.NET Core)
 Windows Presentation Foundation client application
- Revit Extension WPF
 Revit Extension WPF

Not finding what you're looking for?
[Install more tools and features](#)

[Next](#)

Na tela que segue deveremos escolher o nome do projeto e da solução que iremos construir, bem como o caminho onde armazenaremos nossa solução. Escolhidas as opções, clicamos em “Create” para criar nossa solução.

Configure your new project

TL Revit 2020 WPF Addin C# WPF Revit Add-In

Project name

TL Revit 2020 WPF Addin1

Location

D:\Documents\J3di\Visual Studio\Source\Repos

...

Solution name ⓘ

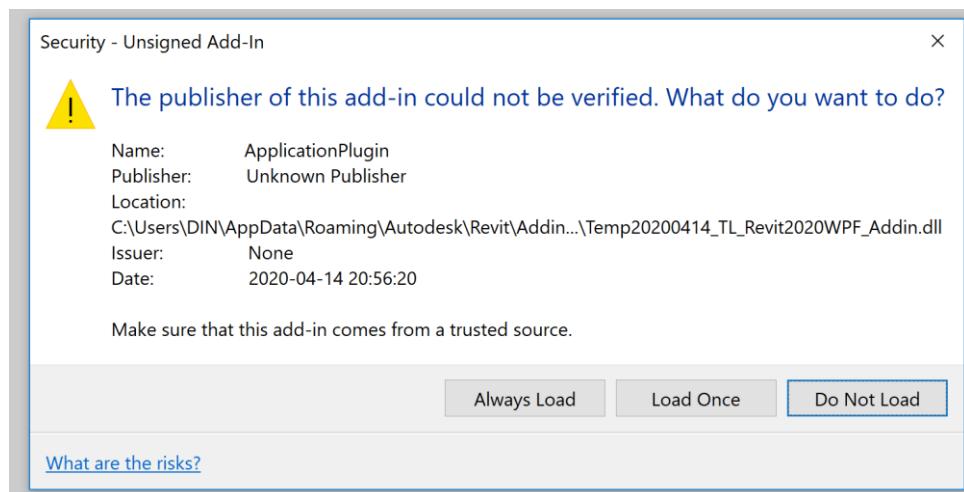
TL Revit 2020 WPF Addin1

Place solution and project in the same directory

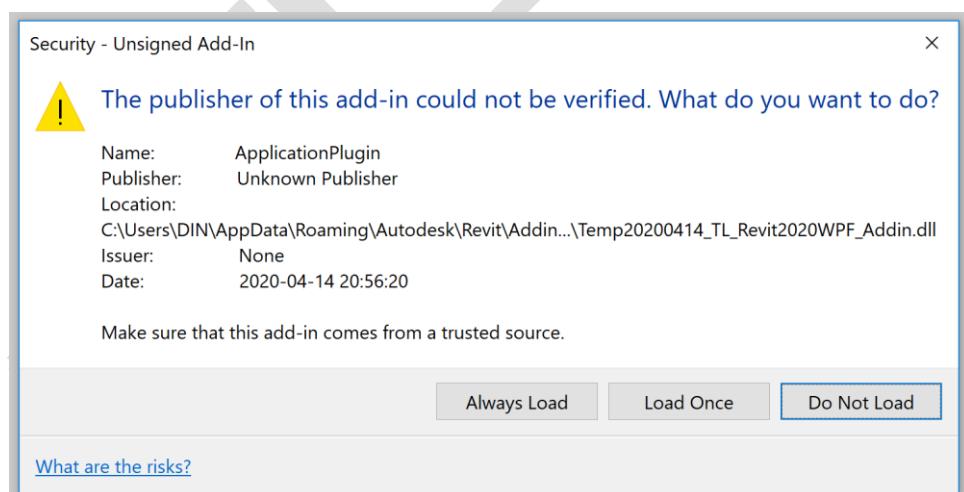
[Back](#)

[Create](#)

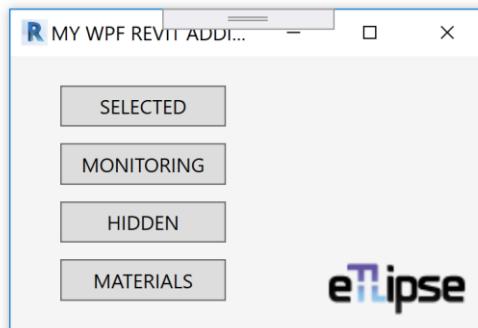
O projeto está criado e será aberto para edição. Este template traz um exemplo a partir do qual é possível fazer as devidas personalizações para a solução que desejar criar. O código é amplamente comentado para que possa facilitar sua edição.



Uma vez que tenha construído sua aplicação sobre o template, poderá então compilar sua solução. O projeto irá abrir o Revit e lanças sua solução. Importante notar que, por proteção do Revit, será exibida a tela inicial para autorizar o uso da aplicação, que poderá ser apenas para uma vez, para sempre, ou pode não permitir o carregamento. Finalmente pode abrir um projeto no Revit e a solução criada estará no menu que terá o nome configurado na solução.



O ícone projetado estará disponível para lançar sua solução. A janela projetada será lançada ao clicar no ícone do seu aplicativo.



eTlipse

Exemplos Iniciais Utilizando a API do Revit

Utilizando as informações sobre o uso da API do Revit, com C# no Visual Studio, podemos estudar alguns exemplos para visualizar o uso da API. Estes primeiros exemplos visam auxiliar o entendimento das funcionalidades mais gerais, que precisaremos entender na sequência dos estudos, além de promover experiências que encorajam a continuidade do estudo apresentando resultados práticos.

Exemplo - Coletando Informação de Elementos Selecionados

É possível obter informações de elementos previamente selecionados. Esta possibilidade nos permite pesquisar parâmetros dos elementos e guarda-los para obter uma base de dados. Criamos uma aplicação de comando, tal como anteriormente, e usamos o código a seguir para obter a lista de IDs dos elementos selecionados.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.Attributes;

namespace RvtAPIG14_RetrieveSelElement_20180711{

    [Transaction(TransactionMode.Manual)]
    public class Document_Selection : IExternalCommand {
        Result IExternalCommand.Execute(ExternalCommandData commandData, ref string
message, ElementSet elements) {
            try {
                // Select some element in Revit before invoking this command.

                // Get the handle of the current document.
                UIDocument uiDoc = commandData.Application.ActiveUIDocument;
                Document currentDoc = uiDoc.Document;

                // Get the element selection of current document.
                Selection selection = uiDoc.Selection;
                ICollection<ElementId> selectedIds = selection.GetElementIds();

                if (selectedIds.Count == 0) {
                    // If no elements selected.
                    TaskDialog.Show("Revit", "You haven't selected any elements.");
                } else {
                    String info = "Ids of selected elements in the document are: ";
                    foreach (ElementId elemId in selectedIds) {
                        info += "\n\t" + elemId.IntegerValue;
                    }
                    TaskDialog.Show("Revit", info);
                }
            }
        }
    }
}

```

```

        }

    } catch (Exception e) {
        message = e.Message;
        return Result.Failed;
    }
    return Result.Succeeded;
}
}

}

```

Existem formas diversas de seleção, para que seja possível obter dados de elementos. Alternativamente o código a seguir apresenta outra possibilidade.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace WpfAppRevitAPI20180619 {

    [Transaction(TransactionMode.Manual)]
    class PickElement : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            Element selectedElement =
            doc.GetElement(sel.PickObject(ObjectType.Element, "Select object!"));

            string elementInfo = "Category name: " + selectedElement.Category.Name +
            "\nName: " + selectedElement.Name;

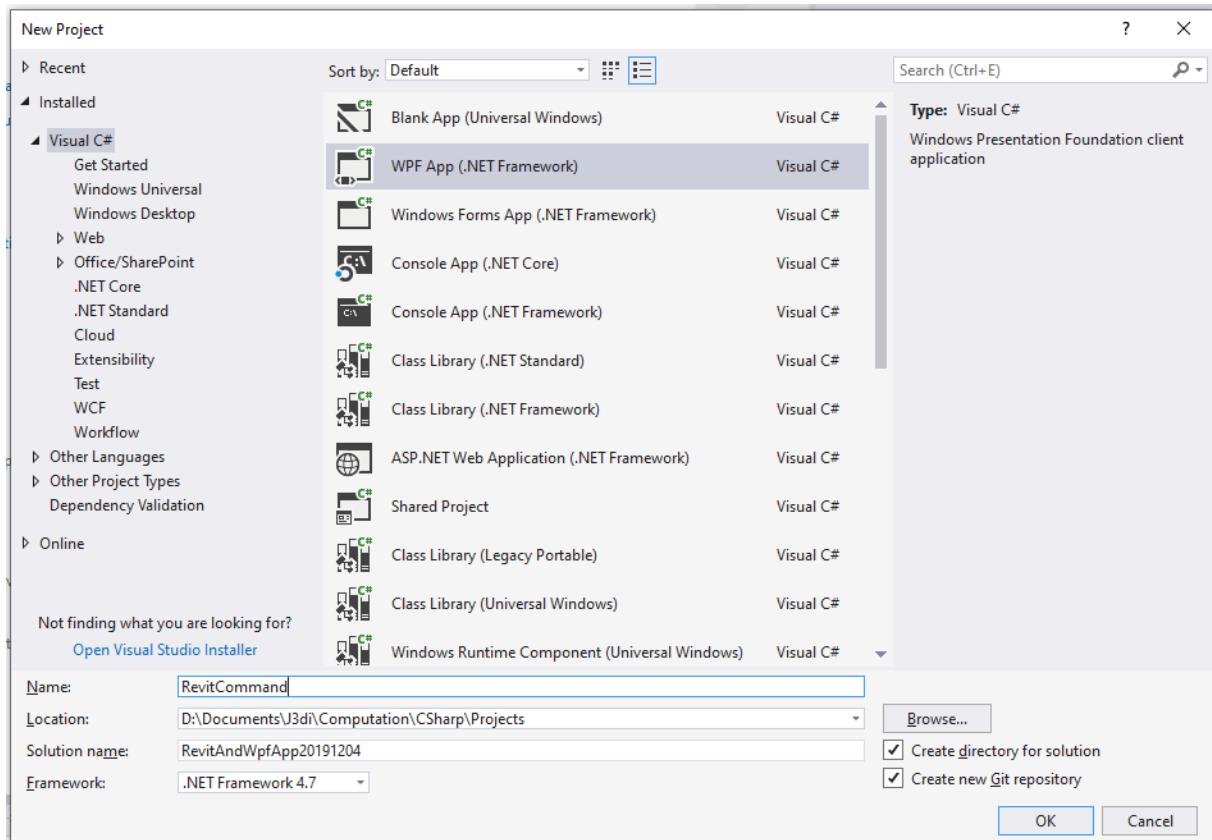
            TaskDialog.Show("Object information: ", elementInfo);

            return Result.Succeeded;
        }
    }
}

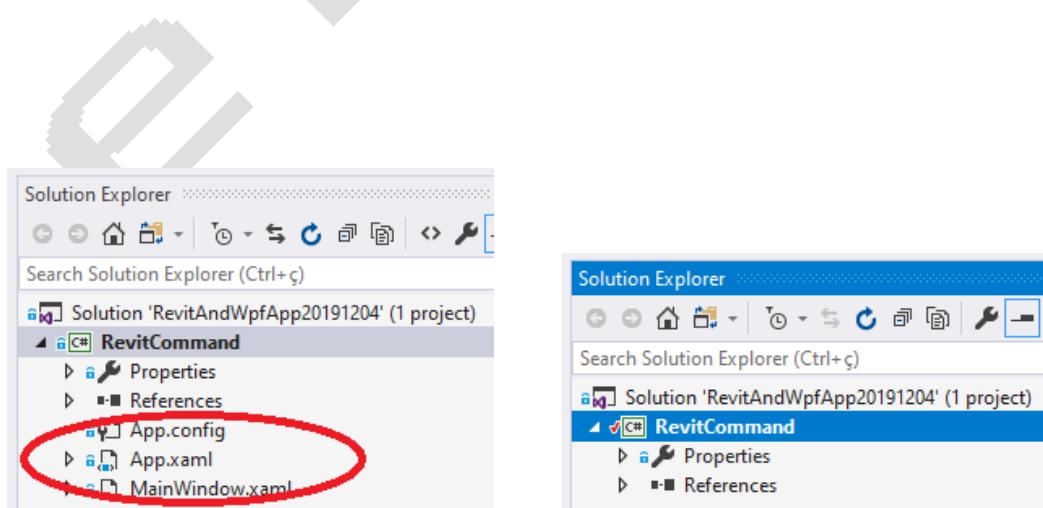
```

Exemplo - Revit com WPF

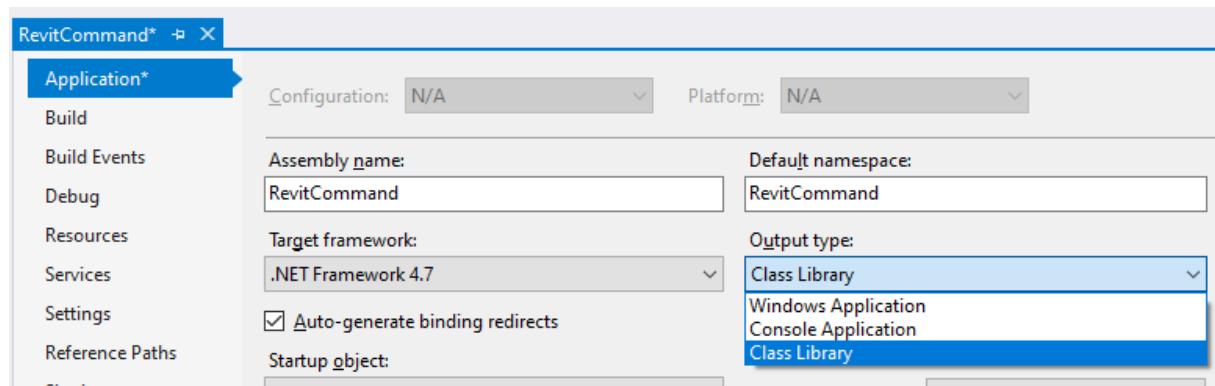
É, conforme já comentado, bastante interessante o uso de WPF em soluções Windows, sendo uma solução bem eficiente comparada a WindowsForm. Neste exemplo é possível ver o uso do WPF a partir da API do Revit. Iniciaremos com um novo projeto no Visual Studio, mas desta vez optando por WPF, conforme segue.



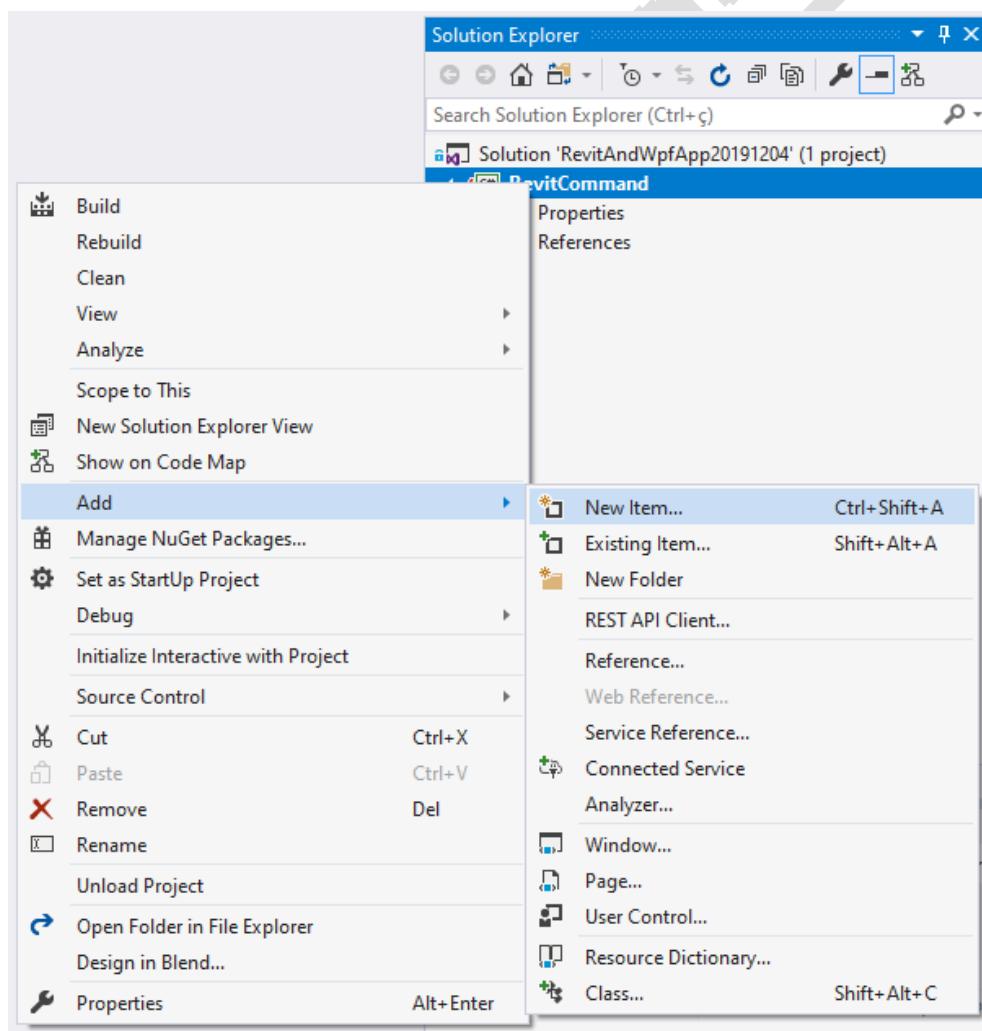
O Visual Studio cria por padrão uma janela WPF e seus arquivos de suporte, que não precisamos necessariamente utilizar em nossa aplicação. Desta forma, podemos apagar App.config, App.xaml e MainWindow.xaml.

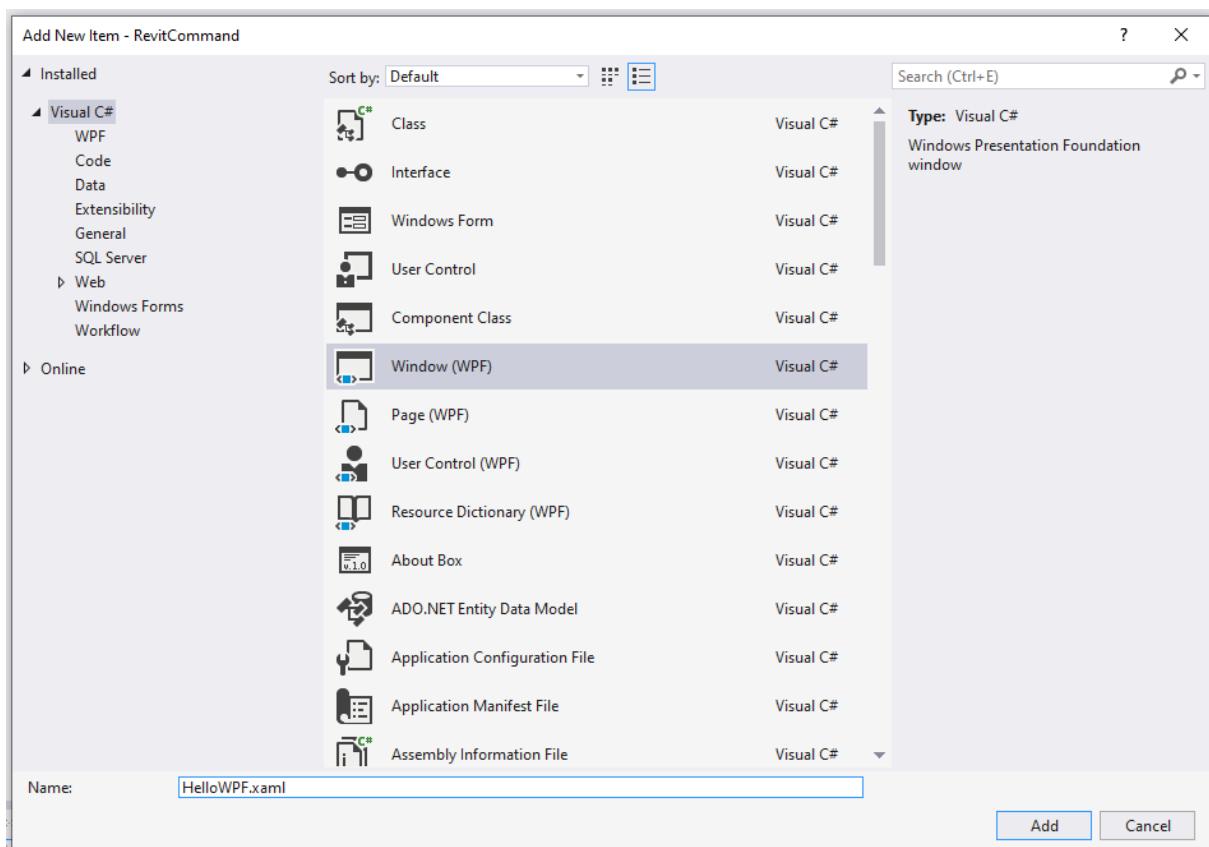


Precisamos acessar as propriedades do projeto (Menu Project-><Project> Properties...) e, na guia *Application*, mudar o *Output Type* de *Windows Application* para *Class Library*.



Para didaticamente mostrar que podemos criar nossas próprias janelas, deletamos a janela WPF criada automaticamente por padrão, e agora adicionaremos uma nova ao projeto. Botão direito no projeto, no Solution Explorer, e podemos optar por adicionar novo item. Em seguida optamos por janela WPF.





Ajustaremos o código XAML de acordo com o código a seguir, a título de exemplo, sendo possível padronizar esta janela.

```

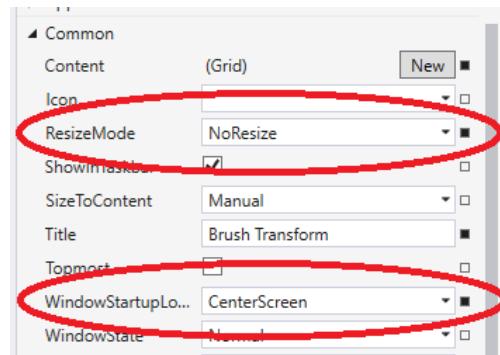
<Window x:Class="RevitCommand.HelloWPF"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:RevitCommand"
    mc:Ignorable="d"
    Title="Hello WPF" Height="300" Width="300">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button x:Name="btnMessage" Width="100" Height="33" Margin="5, 9, 5, 5"
Click="btnMessage_Click">Message</Button>
    </StackPanel>
</Window>
<Window x:Class="RevitCommand.HelloWPF"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:RevitCommand"
    mc:Ignorable="d"
    Title="Hello WPF" Height="300" Width="300">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <Button x:Name="btnMessage" Width="100" Height="33" Margin="5, 9, 5, 5"
Click="btnMessage_Click">Message</Button>
    </StackPanel>
</Window>

```

Boas práticas:

```
/*
```

Uma ação que pode ser interessante, para usar WPF com a API do Revit, é evitar que a janela seja minizada/maximizada e ser colocada no centro da janela.



```
*/
```

Uma implementação necessária, quando usar WPF e a API do Revit, é tornar a Janela “*disposable*”, visto não ser o default do WPF.

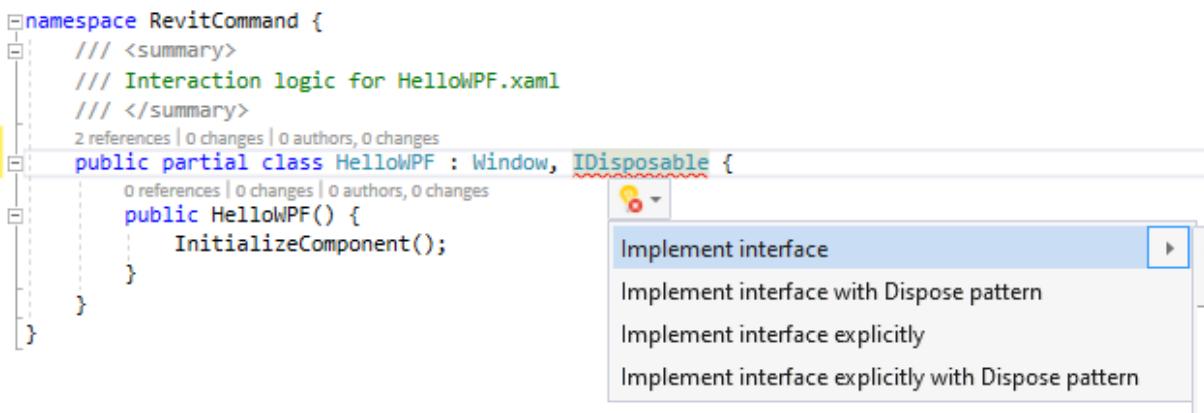
Em HelloWPF.xaml.cs devemos ter:

```
public partial class mainWindow : Window, IDisposable
```

Precisamos então implementar a interface *IDisposable* e inserir o seguinte método:

```
public void Dispose() {
    this.Close();
}
```

O Visual Stúdio auxilia na implementação de Dispose quando indicamos que a janela WPF implementa *IDisposable*.



O uso da diretiva `using()`, por exemplo, permitirá o uso deste método para operação segura com a API. O código ficará (já incluindo o clique do botão adicionado na janela WPF) como segue.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

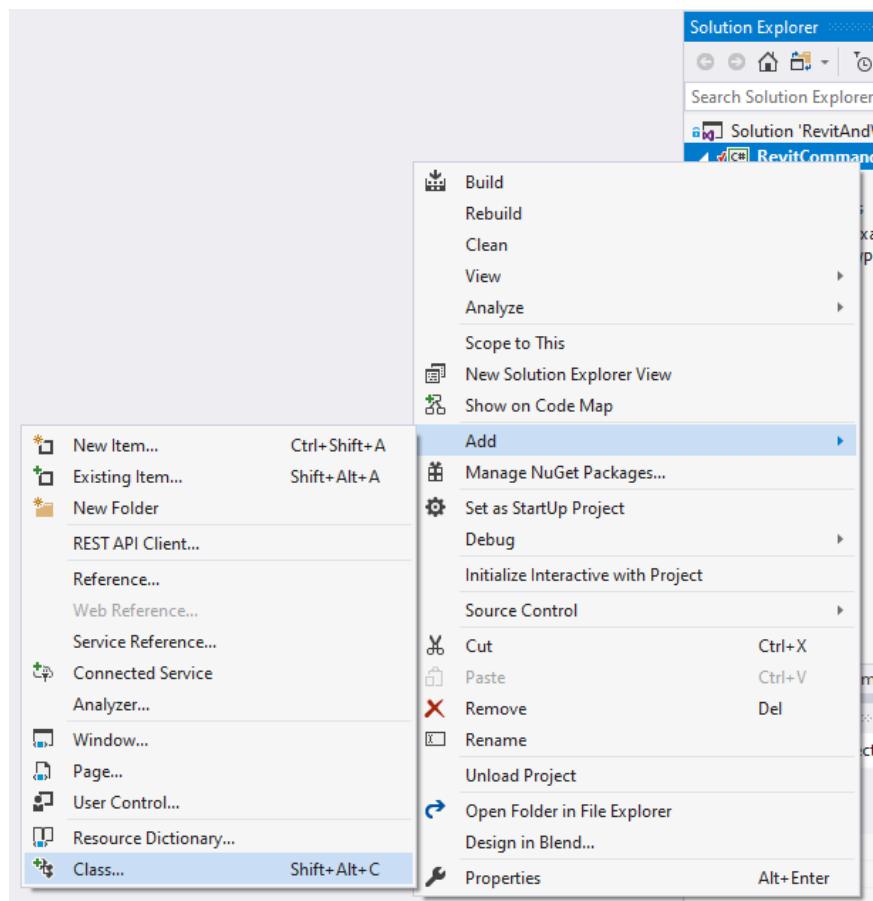
namespace RevitCommand {
    /// <summary>
    /// Interaction logic for HelloWPF.xaml
    /// </summary>
    public partial class HelloWPF : Window, IDisposable {
        public HelloWPF() {
            InitializeComponent();
        }

        public void Dispose() {
            this.Close();
        }

        private void btnMessage_Click(object sender, RoutedEventArgs e) {
            MessageBox.Show("Revit API by eTlipse!", "Ola Revit com WPF!");
        }
    }
}

```

Iremos adicionar uma nova classe para implementar o comando da API do Revit. Dentro do comando instanciamos a janela WPF e a exibimos a partir do Revit. O código ficará como ilustrado na sequência.



Devemos seguir o procedimento usual para comando com a API do Revit, já descrito, como a adição das bibliotecas necessárias. O código ficará assim:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {
            HelloWPF helloWPF = new HelloWPF();
            helloWPF.Show();

            return Result.Succeeded;
        }
    }
}

```

}

Exemplo - Adoção de *Using* e *Transaction*

Programadores de C# devem estar acostumados com recursos como coleta de “lixo” automática, mas o uso da API do Revit requer cuidados especiais. Uma boa prática consiste em encapsular rotinas em bloco “using” para uma dada transação. O código a seguir exemplifica essa prática.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace WpfAppRevitAPI20180619 {

    [Transaction(TransactionMode.Manual)]
    class MoveElement : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            Element selectedElement =
                doc.GetElement(sel.PickObject(ObjectType.Element, "Select element like column!"));

            LocationPoint selectedElementLocation = selectedElement.Location as
            LocationPoint;

            // Tip 1: CTRL + K + S for involve the code to dispose! Than using!
            // Tip 2: CTRL + K + D to organize the code!

            using (Transaction trn = new Transaction(doc, "Move element like
            column!")) {
                trn.Start();
                selectedElementLocation.Move(XYZ.BasisX.Multiply(5));
                trn.Commit();
            }
            return Result.Succeeded;
        }
    }
}

```

Exemplo - Determinando Valor de Parâmetro de Objeto Selecionado

É possível “setar”, determinar o valor de um dado parâmetro de um objeto. Localizamos o parâmetro desejado usando seu nome e depois podemos alterar seu valor. Criamos uma aplicação de comando, tal como anteriormente, e usamos o código a seguir para alterar o valor de um parâmetro do elemento selecionado.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;

namespace BIMRevitAPI20181017 {

    [Transaction(TransactionMode.Manual)]
    class BIMRevitAPI_20181017 : IExternalCommand {

        // Plugin entrance
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            try {
                // Connecting Application, Document, etc...
                UIApplication uiApp = commandData.Application;
                UIDocument uiDoc = uiApp.ActiveUIDocument;
                Application app = uiApp.Application;
                Document doc = uiDoc.Document;
                Selection sel = uiDoc.Selection;

                Element e = SelectElement(uiDoc, doc);
                Parameter parameter = e.LookupParameter("Comentários");

                using (Transaction t = new Transaction(doc, "Parameter")) {
                    t.Start("Param");
                    try {
                        parameter.Set("Comments Column");
                    } catch (Exception ex) {
                        message = ex.Message;
                        TaskDialog.Show("Error!", message);
                    }
                    t.Commit();
                }

            } catch (Exception ex) {
                message = ex.Message;
                TaskDialog.Show("Error!", message);
                return Result.Failed;
            }
            return Result.Succeeded;
        }
    }
}

```

```

// Select Element
public Element SelectElement(UIDocument uiDoc, Document doc) {
    Reference reference = uiDoc.Selection.PickObject(ObjectType.Element);
    Element element = uiDoc.Document.GetElement(reference);
    return element;
}

// Get Parameter Value
public string GetParameterValue(Parameter parameter) {
    switch (parameter.StorageType) {
        case StorageType.None:
            return parameter.AsValueString();
        case StorageType.Integer:
            // Get value with unit, AsInteger() can get value without unit
            return parameter.AsValueString();
        case StorageType.Double:
            // Get value with unit, AsDouble() can get value without unit
            return parameter.AsValueString();
        case StorageType.String:
            return parameter.AsValueString();
        case StorageType.ElementId:
            return parameter.AsElementId().IntegerValue.ToString();
        default:
            return "";
    }
}
}

```

Exemplo - Obtendo Valor de Parâmetro de Objeto Selecionado

É possível igualmente obter o valor de um dado parâmetro de um objeto. Localizamos o parâmetro desejado usando seu nome e depois podemos resgatar seu valor. Criamos uma aplicação de comando, tal como anteriormente, e usamos o código a seguir para alterar o valor de um parâmetro do elemento selecionado.

// Atenção - Revit trabalha internamente com pés (feet). 1 feet = 0,3048 meters!

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;

```

```

namespace BIMRevitAPI20181017 {

    [Transaction(TransactionMode.Manual)]
    class BIMRevitAPI_20181017 : IExternalCommand {

        // Plugin entrance
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            try {
                // Connecting Application, Document, etc...
                UIApplication uiApp = commandData.Application;
                UIDocument uiDoc = uiApp.ActiveUIDocument;
                Application app = uiApp.Application;
                Document doc = uiDoc.Document;
                Selection sel = uiDoc.Selection;

                Element e = SelectElement(uiDoc, doc);
                Parameter parameter =
e.get_Parameter(BuiltInParameter.WALL_BASE_OFFSET);

                using (Transaction t = new Transaction(doc, "Param")) {
                    t.Start("Param");
                    try {
                        parameter.Set(-5); // 5 feet - Revit trabalha internamente com
feet - 1 feet = 0,3048 meters
                    } catch (Exception ex) {
                        message = ex.Message;
                        TaskDialog.Show("Error!", message);
                    }
                    t.Commit();
                    TaskDialog.Show("New Value", GetParameterValue(parameter));
                }

                } catch (Exception ex) {
                    message = ex.Message;
                    TaskDialog.Show("Error!", message);
                    return ResultFailed;
                }
                return Result.Succeeded;
            }

            // Select Element
            public Element SelectElement(UIDocument uiDoc, Document doc) {
                Reference reference = uiDoc.Selection.PickObject(ObjectType.Element);
                Element element = uiDoc.Document.GetElement(reference);
                return element;
            }

            // Get Parameter Value
            public string GetParameterValue(Parameter parameter) {
                switch (parameter.StorageType) {
                    case StorageType.None:
                        return parameter.AsValueString();
                    case StorageType.Integer:
                        // Get value with unit, AsInteger() can get value without unit
                        return parameter.AsValueString();
                    case StorageType.Double:
                        // Get value with unit, AsDouble() can get value without unit
                        return parameter.AsValueString();
                    case StorageType.String:
                }
            }
        }
    }
}

```

```
        return parameter.AsValueString();
    case StorageType.ElementId:
        return parameter.AsElementId().IntegerValue.ToString();
    default:
        return "";
    }
}
}
```

Exemplo - Exportando para Excel

É possível exportar informações do Revit para o Excel, esta pode ser uma funcionalidade bem interessante para gerar uma base de dados simples que pode facilmente ser manipulada pelos usuários. Criamos uma aplicação de comando, tal como usual, e usamos o código a seguir para gerar um arquivo de saída no Excel.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Excel = Microsoft.Office.Interop.Excel;
using System.Windows;
using System.Runtime.InteropServices;

namespace RevitToExcel20190417 {

    [Transaction(TransactionMode.Manual)]
    class RevitToExcel20190417 : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {
            try {
                UIApplication uiApp = commandData.Application;
                UIDocument uiDoc = uiApp.ActiveUIDocument;
                Autodesk.Revit.ApplicationServices.Application app =
uiApp.Application;
                Document doc = uiDoc.Document;
                Selection sel = uiDoc.Selection;

                ICollection<ElementId> elementIds = sel.GetElementIds();

                using (Transaction t = new Transaction(doc, "IDs")) {
                    if (elementIds.Count == 0) {
                        // If no element selected
                        TaskDialog.Show("Revit", "You haven't selected any element!");
                    }
                }
            }
        }
    }
}
```

```
        } else {
            // Interação com o Excel
            Excel.Application xlApp = new
Microsoft.Office.Interop.Excel.Application();
            if (xlApp == null) {
                MessageBox.Show("Excel is not properly installed!!");
                return Result.Failed;
            }
            Excel.Workbook xlWorkBook;
            Excel.Worksheet xlWorkSheet;
            object misValue = System.Reflection.Missing.Value;
            xlWorkBook = xlApp.Workbooks.Add(misValue);
            xlWorkSheet =
(Excel.Worksheet)xlWorkBook.Worksheets.get_Item(1);

            // Interação com o Revit
            String info = "IDs of selected elements in the document are:";
            int x = 1;
            int y = 1;
            xlWorkSheet.Cells[x, y] = info;
            x = x + 1;
            foreach (ElementId item in elementIds) {
                //info += "\n\t" + item.IntegerValue;
                xlWorkSheet.Cells[x, 1] = item.IntegerValue;
                x = x + 1;
            }
            // TaskDialog.Show("Revit", info);

            // Saving Excel File
            xlWorkBook.SaveAs("d:\\temp\\Revit-Excel.xls",
Excel.XlFileFormat.xlWorkbookNormal, misValue, misValue, misValue, misValue,
Excel.XlSaveAsAccessMode.xlExclusive, misValue, misValue,
misValue, misValue, misValue);
            xlWorkBook.Close(true, misValue, misValue);
            xlApp.Quit();

            // Cuidando da memória!
            Marshal.ReleaseComObject(xlWorkSheet);
            Marshal.ReleaseComObject(xlWorkBook);
            Marshal.ReleaseComObject(xlApp);

            MessageBox.Show("Excel file created, you can find the file
d:\\temp\\Revit-Excel.xls");
        }
    }

    return Result.Succeeded;
}

} catch (Exception ex) {
    TaskDialog.Show("Error!", ex.Message);
    return Result.Failed;
}
}

}
```

Uso de Filtros e Técnicas para Seleção de Elementos

O uso de filtros específicos nos permitem selecionar um grupo específico de elementos baseados em dados critérios. Esta é uma opção fundamental para que possamos operar apenas em um grupo restrito de elementos que correspondam às nossas necessidades.

Exemplo - Filtrando Elementos

Podemos filtrar os elementos para trabalhar apenas com uma dada categoria. Criamos uma aplicação de comando, tal como usual, e usamos o código a seguir para obter informações apenas do nome de paredes instanciadas.

```
// Usando o atalho WhereElementIsNotElementType() nos permite encontrar apenas
// instâncias de determinada categoria.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace Temp20190501 {

    [Transaction(TransactionMode.Manual)]
    class Temp20190501 : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {
            // Estabelecendo conexões.
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Find all wall instances in the document by using category filter.
                ElementCategoryFilter filter = new
                ElementCategoryFilter(BuiltInCategory.OST_Walls);
                // Apply the filter to the elements in the active document.
                // Use shortcut WhereElementIsNotElementType() to find wall instances
                // only.
                FilteredElementCollector collector = new
                FilteredElementCollector(doc);
                IList<Element> walls =
                collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
                String prompt = "The walls in the current document are:\n";
                foreach (Element e in walls) {
                    prompt = prompt + e.Name + "\n";
                }
            }
        }
    }
}
```

```
        }
        TaskDialog.Show("Revit!", prompt);
        return Result.Succeeded;
    } catch (Exception ex) {
        // In case of error.
        TaskDialog.Show("Error!", ex.Message);
        return Result Failed;
    }
}
}
}
```

Exemplo - Usando Filtro de Parâmetro para Selecionar Room pela Área

Podemos filtrar os elementos por dado parâmetro, neste exemplo para selecionar room com área superior e menor ou igual a dada valor. Criamos uma aplicação de comando, tal como usual, e usamos o código a seguir para obter informações dos rooms segundo sua área.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace Temp20190501 {

    [Transaction(TransactionMode.Manual)]
    class Temp20190501 : IExternalCommand {
        /// <summary>
        /// Comando para Revit usando sua API.
        /// </summary>
        /// <param name="commandData"></param>
        /// <param name="message"></param>
        /// <param name="elements"></param>
        /// <returns></returns>
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {
            // Estabelecendo conexoes.
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                /* Criar um filtro ElementParameter para encontrar rooms com área
superiores a um dado valor
                 * Criar um filtro usando provider e evaluator
                 */
                BuiltInParameter areaParam = BuiltInParameter.ROOM_AREA;
```

```
// Provider
ParameterValueProvider pvp = new ParameterValueProvider(new
ElementId((int)areaParam));
// Evaluator
FilterNumericRuleEvaluator fnrv = new FilterNumericGreater();
// Rule value
double ruleValue = 100.0f; // Filtrar rooms com area superior a este
valor.
// Rule
FilterRule fRule = new FilterDoubleRule(pvp, fnrv, ruleValue, 1E-6);

// Criar um filtro ElementParameter
ElementParameterFilter filter = new ElementParameterFilter(fRule);

// Aplicar o filtro para os elementos do documento ativo.
FilteredElementCollector collector = new
FilteredElementCollector(doc);
IList<Element> rooms = collector.WherePasses(filter).ToElements();
String prompt = "Rooms com área > 100 no documento ativo:\n";
foreach (Element element in rooms) {
    prompt = prompt + element.Name + "\n";
}

// Encontrar rooms com areas menor que ou igual a 100.
fnrv = new FilterNumericLessOrEqual();
fRule = new FilterDoubleRule(pvp, fnrv, ruleValue, 1E-6);
ElementParameterFilter lessOrEqualFilter = new
ElementParameterFilter(fRule, false);
collector = new FilteredElementCollector(doc);
IList<Element> lessOrEqualFounds =
collector.WherePasses(lessOrEqualFilter).ToElements();
prompt = prompt + "\n Rooms com área <= 100 no documento ativo:\n";
foreach (Element element in lessOrEqualFounds) {
    prompt = prompt + element.Name + "\n";
}
TaskDialog.Show("Revit API!", prompt);

return Result.Succeeded;
} catch (Exception ex) {
    // Em caso de erro ...
    TaskDialog.Show("Error!", ex.Message);
    return ResultFailed;
}
}
}
}
```

Exemplo - Combinando Filtros para Selecionar Portas Instanciadas

Podemos usar uma combinação de filtros, e agrupa-los logicamente de forma que nossa seleção obedeça a mais de um critério como filtro. Neste exemplo usaremos um filtro para selecionar instâncias e que sejam da categoria portas.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexoes
           UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Encontrar todas as instancias de portas no projeto encontrando
                todos os elementos que
                // pertencem a categoria porta e sao familia de instancia
                ElementClassFilter familyInstanceFilter = new
                ElementClassFilter(typeof(FamilyInstance));

                // Criar uma categoria filtro para portas
                ElementCategoryFilter doorsCategoryFilter = new
                ElementCategoryFilter(BuiltInCategory.OST_Doors);

                // Criar a logica e filtro para Door FamilyInstance
                LogicalAndFilter doorInstancesFilter = new
                LogicalAndFilter(familyInstanceFilter, doorsCategoryFilter);

                // Aplicar o filtro para os elementos ativos no documento
                FilteredElementCollector collector = new
                FilteredElementCollector(doc);
                IList<Element> doors =
                collector.WherePasses(doorInstancesFilter).ToElements();

                // Listar as portas
                String prompt = "Portas:\n";
                foreach (Element element in doors) {
                    prompt = prompt + "ID - " + element.Id + " / Nome - " +
                    element.Name + "\n";
                }
                TaskDialog.Show("Revit API", prompt);

                return Result.Succeeded;
            } catch (Exception ex) {
                message = ex.Message;
                TaskDialog.Show("Error!", message);
                return ResultFailed;
            }
        }
    }
}

```

}

Exemplo - Usando o Padrão Interator

Um importante padrão para interagir com uma coleção de dados é o padrão *interator*. Este exemplo mostra como usar este padrão.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.DB.Architecture;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexões
           UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Usar filtro para encontrar todos os rooms do documento
                RoomFilter filter = new RoomFilter();

                // Aplicar o filtro para os elementos no documento ativo
                FilteredElementCollector collector = new
FilteredElementCollector(doc);
                collector.WherePasses(filter);

                // Obter resultado como ElementId Interator
                FilteredElementIdIterator roomIdItr =
collector.GetElementIdIterator();
                roomIdItr.Reset();
                while (roomIdItr.MoveNext()){
                    ElementId roomId = roomIdItr.Current;
                    // Quando o room for menor que 50
                    Room room = doc.GetElement(roomId) as Room;
                    if (room.Area < 50.0) {
                        String prompt = "Room e muito pequeno: id = " +
roomId.ToString();
                    }
                }
            }
        }
    }
}

```

```
        TaskDialog.Show("Revit API", prompt);
        break;
    }
}
return Result.Succeeded;
} catch (Exception ex) {
    message = ex.Message;
    TaskDialog.Show("Error!", message);
    return Result.Failed;
}
}
}
}
```

Exemplo - Usando LINQ

Um dos recursos mais interessantes para interagir com conjunto de dados usando o C# é o LINQ, Language Integrated Query. Este exemplo mostra como utilizar este poderoso recurso com a API do Revit.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo as conexoes
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Usar ElementClassFilter para encontrar instancias de familias cujo
                nome seja 10" Coluna
                ElementClassFilter filter = new
                ElementClassFilter(typeof(FamilyInstance));

                // Aplicar o filtro para os eleentos no documento ativo
            }
        }
    }
}
```

```

        FilteredElementCollector collector = new
FilteredElementCollector(doc);
        collector.WherePasses(filter);

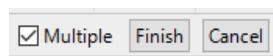
        // Usar Linq para encontrar a instancia de familia cujo nome seja 10"
Coluna
        var query = from element in collector
                    where element.Name == "10\" Coluna"
                    select element;
        // Fazer um cast do elemento encontrado para instancia de familia
        // essa conversao é segura para FamilyInstance porque foi usado
ElementClassFilter para FamilyInsance
        List<FamilyInstance> familyInstances =
query.Cast<FamilyInstance>().ToList<FamilyInstance>();
        String prompt = "Faily Instances:\n";
        foreach (FamilyInstance instance in familyInstances) {
            prompt = prompt + instance.Id.ToString() + "\n";
        }
        TaskDialog.Show("Revit API", prompt);
        return Result.Succeeded;
    } catch (Exception ex) {
        message = ex.Message;
        TaskDialog.Show("Error!", message);
        return ResultFailed;
    }
}
}

}

```

Exemplo - Criando um Novo Grupo a Partir de uma Seleção

Uma vez que tenhamos uma seleção, é possível gerar um novo grupo de elementos que obedeça a novos requisitos. Neste exemplo fazemos uma seleção de elementos, e posteriormente criamos uma segunda seleção que contenha apenas paredes deste primeiro grupo. Importante que a seleção inicial deve ser finalizada ou cancelada, com o uso dos botões mostrados abaixo, para que a seleção seja computada, no caso de cancelamento a segunda seleção automática não poderá ser realizada.



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

```

```

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Pega os elementos selecionados no documento atual
                IList<Reference> references = sel.PickObjects(ObjectType.Element);

                // Mostra quantidade de elementos selecionados. Eh preciso acionar os
                botoes Finish ou Cancel para finalizar a seleção
                TaskDialog.Show("Revit API!", "Número de elementos selecionados: " +
references.Count.ToString());

                // Criar uma nova seleção de elementos para receber os elementos
                // específicos de interesse da seleção mais ampla
                IList<Element> collection = new List<Element>();

                // Adicionamos apenas Walls aa coleção
                foreach (Reference reference in references) {
                    Element element = doc.GetElement(reference);
                    if (element is Wall) {
                        collection.Add(element);
                    }
                }

                // Informa o resultado da seleção
                if (collection.Count != 0) {
                    TaskDialog.Show("Revit API!", "Total de paredes selecionadas " +
collection.Count.ToString());
                } else {
                    TaskDialog.Show("Revit API!", "Nenhuma parede selecionada!");
                }

                return Result.Succeeded;
            } catch (Exception ex) {
                message = ex.Message;
                TaskDialog.Show("Error!", message);
            }

            return Result.Failed;
        }
    }
}

```

Ainda sobre as possibilidades de uso de seleção podemos analisar o exemplo a seguir, considerando uma seleção prévia.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Selecione alguns elementos antes da execução

                ICollection<ElementId> selectedIds = sel.GetElementIds();

                if (0 == selectedIds.Count) {
                    // Se nenhum elemento for selecionado
                    TaskDialog.Show("Revit", "Você não selecionou nenhum elemento.");
                } else {
                    String info = "Os IDs dos elementos selecionados no documento são:
";
                    foreach (ElementId id in selectedIds) {
                        info += "\n\t" + id.IntegerValue;
                    }
                    TaskDialog.Show("Revit API", info);
                }
            } catch (Exception ex) {
                message = ex.Message;
                TaskDialog.Show("Error!", message);
                return ResultFailed;
            }
            return Result.Succeeded;
        }
    }
}

```

Considerando estas funcionalidades de seleção, podemos alternativamente abordar nosso exemplo de refinar uma seleção da forma que segue.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexoes
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Adquirindo os elementos selecionados no documento corrente
                ICollection<ElementId> selectedIds = sel.GetElementIds();

                // Exibe o numero corrente de elementos selecionados
                TaskDialog.Show("Revit API!", "Numero de elementos selecionados: " +
selectedIds.Count.ToString());

                // Entrar na selecao e filtrar apenas as paredes
                ICollection<ElementId> selectedWallIds = new List<ElementId>();

                foreach (ElementId id in selectedIds) {
                    Element element = doc.GetElement(id);
                    if (element is Wall) {
                        selectedWallIds.Add(id);
                    }
                }

                // Estabelece o conjunto de elementos criados como o conjunto de
                // selecao corrente
                sel.SetElementIds(selectedWallIds);

                // Apresenta algumas informacoes ao usuario
                if (0 != selectedWallIds.Count) {
                    TaskDialog.Show("Revit API!", selectedWallIds.Count.ToString() + " "
Paredes foram selecionadas!");
                } else {
                    TaskDialog.Show("Revit API!", "Nenhuma parede foi selecionada!");
                }

            } catch (Exception ex) {
                message = ex.Message;
                TaskDialog.Show("Error!", message);
                return Result.Failed;
            }
            return Result.Succeeded;
        }
    }
}

```

```

    }
}

}

```

Exemplo - Seleção com Interação do Usuário

É possível aguardar a seleção do usuário, usando possibilidade de seleção de único elemento ou múltiplos elementos. No caso, é possível deixar uma mensagem no status do Revit para auxiliar orientar o usuário.



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Seleciona apenas um elemento do Revit
                int selectionCount = 0;
                Reference hasPickOne = sel.PickObject(ObjectType.Element);
                if (hasPickOne != null) {
                    selectionCount = 1;
                    TaskDialog.Show("Revit API!", "Um elemento foi adicionado aa
selecao!");
                }
            }

            // Seleciona elementos do Revit
            IList<Element> hasPickSome = sel.PickElementsByRectangle("Selecao com
retangulo.");
            if (hasPickSome.Count > 0) {
                int newSelectionCount = hasPickSome.Count;
            }
        }
    }
}

```

```

        string prompt = string.Format("{0} elementos foram adicionados aa
selecao!", newSelectionCount - selectionCount);
        TaskDialog.Show("Revit API!", prompt);
    }

    return Result.Succeeded;

} catch (Exception ex) {
    message = ex.Message;
    TaskDialog.Show("Error!", message);

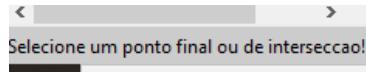
    return Result.Failed;
}
}

}

}

```

O método PickPoint() possui duas sobrecargas com um parâmetro ObjectSnapTypes que permite um ou mais uso de “snap”.



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitAPI20190820 {

    [Transaction(TransactionMode.Manual)]
    class RevitAPI : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexoes
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Snap point
                ObjectSnapTypes snapTypes = ObjectSnapTypes.Endpoints |
ObjectSnapTypes.Intersections;
                XYZ point = sel.PickPoint(snapTypes, "Selecione um ponto final ou de
interseccao!");

                string strCoords = "Ponto selecionado eh " + point.ToString();

                TaskDialog.Show("Revit API!", strCoords);

            } catch (Exception ex) {

```

```

        message = ex.Message;
        TaskDialog.Show("Error!", message);
        return Result.Failed;
    }
    return Result.Succeeded;
}
}
}

```

Exemplo - Coleções e Interações

É bastante útil sabermos interagir com conjuntos de objetos, pois não raro obtemos mais de um resultado ao usar um critério de seleção, dentre outras possibilidades, como poder aplicar uma ação a objetos que possuem uma característica em comum.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Structure;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo conexões
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;
            IList<Element> elems = sel.PickElementsByRectangle();

            try {
                string info = "Elementos selecionados:\n";
                foreach (Element elem in elems) {
                    info += elem.Name + "\n";
                }
                TaskDialog.Show("Revit", info);

                info = "Níveis no documento:\n";
                FilteredElementCollector collector = new
                FilteredElementCollector(doc);
                ICollection<Element> collection =
                collector.OfClass(typeof(Level)).ToElements();
                foreach (Element elem in collection) {
                    // não precisa checar se elem eh nulo

```

```
        info += elem.Name + "\n";
    }
    TaskDialog.Show("Revit", info);

    return Result.Succeeded;
} catch (Exception ex) {
    TaskDialog.Show("Revit API sample fail: ", ex.Message);
    return ResultFailed;
}
}
```

Interagindo com parâmetros

É fundamental utilizar parâmetros no modelo para que tenhamos a metodologia BIM, visto que neles teremos a informação que diferencia um modelo 3D BIM, bem como facilita o ajuste do modelo pela alteração de parâmetros.

Exemplo - Obtendo Parâmetros de Dado Elemento

Obteremos os parâmetros de um elemento selecionado, para posterior exibição em uma janela. O comando permitirá inicialmente a seleção de um único elemento.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RvtAPI_Temp_Prj_20191016 {

    [Transaction(TransactionMode.Manual)]
    public class ParameterInfo : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
```

```

Application app = uiApp.Application;
Document doc = uiDoc.Document;
Selection sel = uiDoc.Selection;

try {
    // Selecionar elemento
    Element element = doc.GetElement(sel.PickObject(ObjectType.Element,
"Selecione um elemento!"));

    // String que recebera os parametros
    StringBuilder st = new StringBuilder();
    st.AppendLine("Mostrar parametros do elemento selecionado:\n");

    // Interagir com os parametros do elemento
    foreach (Parameter para in element.Parameters) {
        st.AppendLine(GetParameterInformation(para, doc));
    }

    // Retornar informacoes
    TaskDialog.Show("Revit API - eTlipse!", st.ToString());

    return Result.Succeeded;
}

} catch (Exception ex) {
    message = ex.Message;
    TaskDialog.Show("Revit API - Error!", message);
    return ResultFailed;
}
}

string GetParameterInformation(Parameter para, Document document) {
    string defName = para.Definition.Name + "\t -> \t";
    // Usar diferentes metodos para adquirir os dados dos parametros de acordo
com o tipo armazenado
    switch (para.StorageType) {
        case StorageType.None:
            defName = "Parametro inesperado!";
            break;
        case StorageType.Integer:
            if (ParameterType.YesNo == para.Definition.ParameterType) {
                if (para.AsInteger() == 0) {
                    defName += " : " + "False";
                } else {
                    defName += " : " + "True";
                }
            } else {
                defName += " : " + para.AsInteger().ToString();
            }
            break;
        case StorageType.Double:
            // Converter o numero em Metro
            defName += para.AsValueString();
            break;
        case StorageType.String:
            defName += " : " + para.AsValueString();
            break;
        case StorageType.ElementId:
            // Encontrar o nome do elemento
            ElementId id = para.AsElementId();
            if (id.IntegerValue >= 0) {
                defName += " : " + document.GetElement(id).Name;
            } else {
                defName += " : " + id.IntegerValue.ToString();
            }
    }
}

```

```
        }
        break;
    default:
        defName = "Parametro inesperado!";
        break;
    }
    return defName;
}
}
```

Exemplo - Obtendo Parâmetros Baseado no Tipo Definido

Desta vez, obteremos parâmetro de um elemento selecionado, baseado na definição de tipo, para posterior exibição em uma janela. O comando permitirá inicialmente a seleção de um único elemento.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RvtAPI_Temp_Prj_20191016 {

    [Transaction(TransactionMode.Manual)]
    public class ParameterInfo : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Selecionar elemento
                Element element = doc.GetElement(sel.PickObject(ObjectType.Element,
"Selecione um elemento!"));

                // String que receberá os dados do parametro
                string prompt = "Parametro :\n";

                // Encontrar parametro
                Parameter elementParameter = FindParameter(element);
                prompt += elementParameter.AsValueString();

                // Exibir informações
                TaskDialog.Show("Revit API - eTlipse!", prompt);
            }
        }
    }
}
```

```
        return Result.Succeeded;

    } catch (Exception ex) {
        message = ex.Message;
        TaskDialog.Show("Revit API - Error!", message);
        return Result.Failed;
    }
}

public Parameter FindParameter(Element element) {
    Parameter foundParameter = null;
    // Encontrar o primeiro parametro que mede comprimento
    foreach (Parameter parameter in element.Parameters) {
        if (parameter.Definition.ParameterType == ParameterType.Length) {
            foundParameter = parameter;
            break;
        }
    }
    return foundParameter;
}
}
```

Exemplo - Obtendo Parâmetros Baseado em *BuiltInParameter*

A manipulação de parâmetros *BuiltInParameter* permite acessar parâmetros padronizados disponíveis como uma enumeração. Este comando deverá ser executado com a seleção de uma parede.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RvtAPI_Temp_Prj_20191016 {

    [Transaction(TransactionMode.Manual)]
    public class ParameterInfo : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo as conexões
            UIApplication uiApp = commandData.Application;
            UIDocument uiDoc = uiApp.ActiveUIDocument;
            Application app = uiApp.Application;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Selecionar parede
                Wall element = doc.GetElement(sel.PickObject(ObjectType.Element,
"Selecione uma parede!")) as Wall;
```

```

        // String que recebera os dados do parametro
        string prompt = "Valor do parametro :\n";

        // Encontrar parametro
        Parameter elementParameter = FindBuiltInParameter(element);
        prompt += elementParameter.AsValueString();

        // Exibir informacoes
        TaskDialog.Show("Revit API - eTlipse!", prompt);

        return Result.Succeeded;

    } catch (Exception ex) {
        message = ex.Message;
        TaskDialog.Show("Revit API - Error!", message);
        return ResultFailed;
    }
}
public Parameter FindBuiltInParameter(Wall wall) {
    // Usar WALL_BASE_OFFSET para acessar o parâmetro base offset
    da parede
    BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
    Parameter parameter = wall.get_Parameter(paraIndex);
    return parameter;
}
}

```

Exemplo - StoreType e Conversão para Unidade Interna

Veremos um exemplo de como modificar um parâmetro baseado no *StoreType*, ou seja, neste exemplo, checaremos se o tipo do parâmetro é um *Double*, caso positivo, poderemos fazer a alteração. Uma vez que haverá alteração do modelo, faremos uso de *Transaction*. Este exemplo também traz uma fundamental funcionalidade, a conversão de valores para unidade interna do Revit.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RvtAPI_Temp_Prj_20191016 {

    [Transaction(TransactionMode.Manual)]
    public class ParameterInfo : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo as conexoes

```

```

UIApplication uiApp = commandData.Application;
UIDocument uiDoc = uiApp.ActiveUIDocument;
Application app = uiApp.Application;
Document doc = uiDoc.Document;
Selection sel = uiDoc.Selection;

try {
    // Selecionar parede
    Wall wall = doc.GetElement(sel.PickObject(ObjectType.Element,
"Selecione uma parede!")) as Wall;
    // Usar WALL_BASE_OFFSET para acessar o parâmetro base
    // offset da parede
    BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
    Parameter parameter = wall.get_Parameter(paraIndex);
    bool valueChanged = false;
    double newValue = 0.3;

    // Parametro original a testar
    TaskDialog.Show("Revit API - eTlipse!", "Valor original do parametro :
\t" + parameter.AsValueString());

    // Fazer o teste e alterar o valor, se teste der verdadeiro
    valueChanged = SetParameter(doc, parameter, newValue);

    if (valueChanged) {
        // Parametro modificado
        TaskDialog.Show("Revit API - eTlipse!", "Novo valor do parametro :
\t" + parameter.AsValueString());
    }
}

return Result.Succeeded;

} catch (Exception ex) {
    message = ex.Message;
    TaskDialog.Show("Revit API - Error!", message);
    return ResultFailed;
}
}

public bool SetParameter(Document doc, Parameter parameter, double value) {
    bool result = false;

    using (Transaction transaction = new Transaction(doc, "Testar
StorageType")) {
        transaction.Start();
        // Se o parametro eh apenas leitura, nao eh possivel alterar o valor
        if (null != parameter && !parameter.IsReadOnly) {
            StorageType parameterType = parameter.StorageType;
            if (StorageType.Double != parameterType) {
                // Os valores sao diferentes e nao devemos alterar o valor
                transaction.Rollback();
                throw new Exception("O tipo gravado e o valor do parametro sao
diferentes!");
            } else {
                // Se nao sao diferentes estabelecemos o resultado para
                // verdadeiro
                // Converter o valor de metro para unidade interna
                result = parameter.Set(UnitUtils.ConvertToInternalUnits(value,
DisplayUnitType.DUT_METERS));
                transaction.Commit();
            }
        }
    }
}

```

```

        }
        return result;
    }

}
}

```

Exemplo - AsValueString e SetValueString

Parâmetros que representa medidas pode possuir uma representação particular, uma formatação especial. Neste exemplo é possível ler o parâmetro na sua formatação específica e definir seu novo valor igualmente nesta representação.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Estabelecendo conexoes
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Selecionar uma parede
                Wall wall = doc.GetElement(sel.PickObject(ObjectType.Element,
"Selecione uma parede!")) as Wall;

                // Acessar o parametro base offset da parede
                BuiltInParameter paramBaseIdx = BuiltInParameter.WALL_BASE_OFFSET;
                Parameter paramBase = wall.get_Parameter(paramBaseIdx);

                // Acessar o parametro area da parede
                BuiltInParameter paraAreaIdx = BuiltInParameter.HOST_AREA_COMPUTED;
                Parameter paramArea = wall.get_Parameter(paraAreaIdx);

                // Recuperando o parametro area como string
                TaskDialog.Show("Revit API sample!", "Area Parameter: " +
paramArea.AsValueString());

                // Definindo novo valor para base offset
                using (Transaction transaction = new Transaction(doc, "Setar novo
valor de base offset."))
                {

```

```

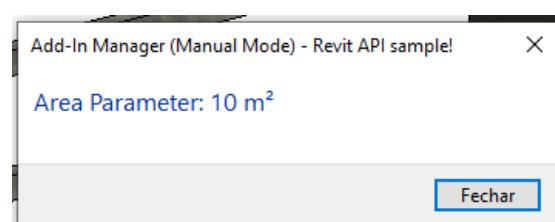
            transaction.Start();
        if (!paramBase.IsReadOnly) {
            paramBase.SetValueString("0.3");
        }
        transaction.Commit();
    }

    return Result.Succeeded;
} catch (Exception ex) {
    TaskDialog.Show("Revit API sample fail: ", ex.Message);
    return ResultFailed;
}
}

}

}

```



Exemplo - Parâmetros Relacionados

Alguns parâmetros são relacionados, de tal forma que a alteração de um implica no recálculo, atualização, do outro. Este é o caso da altura do topo de uma abertura e a altura da sua soleira, conforme exemplo a seguir.



```

using System;
using System.Collections.Generic;

```

```

using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo conexoes
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {
                // Selecionar uma parede
                FamilyInstance opening =
doc.GetElement(sel.PickObject(ObjectType.Element, "Selecione uma abertura!")) as
FamilyInstance;

                // Pegar os parametros da altura original da soleira e do topo da
                // abertura
                Parameter sillPara =
opening.get_Parameter(BuiltInParameter.INSTANCE_SILL_HEIGHT_PARAM);
                Parameter headPara =
opening.get_Parameter(BuiltInParameter.INSTANCE_HEAD_HEIGHT_PARAM);
                double sillHeight = sillPara.AsDouble();
                double origHeadHeight = headPara.AsDouble();

                // Mudando apenas a altura da soleira
                using (Transaction transaction = new Transaction(doc, "Setar novo
                valor de altura da soleira.")) {
                    transaction.Start();
                    sillPara.Set(sillHeight + 2.0);
                    transaction.Commit();
                }

                // A altura do topo eh recalculada
                double newHeadHeight = headPara.AsDouble();
                MessageBox.Show("Antiga altura do topo: " + origHeadHeight + "; nova
                altura do topo: " + newHeadHeight);

                return Result.Succeeded;
            } catch (Exception ex) {
                TaskDialog.Show("Revit API sample fail: ", ex.Message);
                return ResultFailed;
            }
        }
    }
}

```

Interagindo com Elementos

Algumas operações básicas são com frequência executadas de forma repetida, o que traz uma boa oportunidade de otimização com computação.

Exemplo - Movendo Elementos

Uma operação fundamental para possíveis alterações da construção é a operação de mover determinado objeto, esta é uma operação não rara pra BIM quando a detecção de interferência é realizada.

O exemplo que segue demonstra mais de uma possibilidade para realizar este tipo de operação. Inicialmente movemos um pilar pelo modo usual, e posteriormente usamos o recurso de mover seu ponto base. Ainda no mesmo exemplo, usamos dois recursos para reposicionar uma parede, uma usamos a operação de relocação para a parede e posteriormente operamos na sua curva base. Podemos perceber que dependendo do elemento teremos um elemento básico como base, pilar teremos o ponto, parede teremos uma curva, por exemplo.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {

    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {

        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {
```

```

// Estabelecendo conexoes

UIDocument uiDoc = commandData.Application.ActiveUIDocument;
Document doc = uiDoc.Document;
Selection sel = uiDoc.Selection;

try {
    // Selecionar um pilar

    FamilyInstance column =
doc.GetElement(sel.PickObject(ObjectType.Element, "Selecione um pilar!"))
    as FamilyInstance;

    // Adquirir a localizacao atual do pilar

    LocationPoint columnLocation = column.Location as LocationPoint;
    XYZ oldPlace = columnLocation.Point;

    // Usando transacao para mover o pilar

    using (Transaction transactionPilar = new Transaction(doc, "Mover
pilar."))
    {
        transactionPilar.Start();

        // Mover o pilar para um novo local

        XYZ newPlace = new XYZ(5, 10, 10);
        ElementTransformUtils.MoveElement(doc, column.Id, newPlace);
        transactionPilar.Commit();

    }

    // Verificar a nova posicao do pilar

    columnLocation = column.Location as LocationPoint;
    XYZ newActual = columnLocation.Point;

    string info = "Posicao original X: " + oldPlace.X.ToString() + "\nNova
posicao X: " + newActual.X.ToString();

    TaskDialog.Show("Revit API!", info);

    // Selecionar um pilar para mover com Point
}

```

```

        column = doc.GetElement(sel.PickObject(ObjectType.Element, "Seleciona
um pilar.")) as FamilyInstance;

        // Mover pilar usando Point

        using (Transaction transactionPilarPoint = new Transaction(doc, "Mover
pilar com Point."))
        {
            transactionPilarPoint.Start();

            LocationPoint columnPoint = column.Location as LocationPoint;
            if (null != columnPoint)
            {
                XYZ newLocation = new XYZ(10, 20, 0);

                // Mover o pilar para a nova posicao
                columnPoint.Point = newLocation;
            }

            transactionPilarPoint.Commit();

        }

        TaskDialog.Show("Revit API!", "Pilar movido usando Point!");
    }

    // Selecionar uma parede para mover com Location

    Wall wall = doc.GetElement(sel.PickObject(ObjectType.Element,
"Seleciona uma parede com Location.")) as Wall;

    // Mover parede usando Location

    using (Transaction transactionParedeLocation = new Transaction(doc,
"Mover parede."))
    {
        transactionParedeLocation.Start();

        LocationCurve wallLine = wall.Location as LocationCurve;
        XYZ translationVec = new XYZ(5, 15, 0);
        wallLine.Move(translationVec);
        transactionParedeLocation.Commit();

    }

    TaskDialog.Show("Revit API!", "Parede movida usando Location!");

    // Selecionar uma parede para mover com Curve

    wall = doc.GetElement(sel.PickObject(ObjectType.Element, "Seleciona uma
parede.")) as Wall;

```

```
// Mover parede usando Curve

using (Transaction transactionParedeCurve = new Transaction(doc, "Mover
parede com Curve.")) {
    transactionParedeCurve.Start();
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ p1 = XYZ.Zero;
    XYZ p2 = new XYZ(10, 20, 0);
    Line newWallLine = Line.CreateBound(p1, p2);

    // Alterar a linha da parede para a nova linha
    wallLine.Curve = newWallLine;
    transactionParedeCurve.Commit();
}

TaskDialog.Show("Revit API!", "Parede movida usando Curve!");

return Result.Succeeded;

} catch (Exception ex) {
    TaskDialog.Show("Revit API sample fail: ", ex.Message);
    return ResultFailed;
}
}
```

Exemplo - Rotacionando Elementos

Outra operação fundamental para possíveis alterações da construção é a operação de rotacionar determinado objeto. Uma atenção que devemos ter é sobre elementos pinados que, neste caso, não podem ser rotacionados. Caso o elemento possa receber *downcast* para *LocationPoint* ou *LocationCurve*, podemos rotacionar a curva ou o ponto diretamente. Interessante entender se o elemento é alocado baseado em um ponto ou em uma curva, de forma que a opção correta deve ser utilizada para efetiva rotação, por exemplo o comportamento de um pilar é diferente de uma viga.

No exemplo a seguir apresentamos as três opções de rotação, mas comentamos as duas primeiras pois apenas uma por vez é executada, aproveite para ver o funcionamento do template da eTlipse para usar múltiplos comandos. Caso deseje ver os outros comandos em ação neste código, basta comentar o terceiro e deixar sem a marcação de comentário aquele que deseja executar.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Estabelecendo conexões
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                // Selecionar um elemento. Neste caso tipo parede para ter uma linha base!
                Element element = doc.GetElement(sel.PickObject(ObjectType.Element,
                "Selecionar um elemento!"));

                // Rotacionar um pilar baseado em seu eixo
                // RotateColumn(doc, element);

                // Rotacionar elemento baseado em LocationCurve
                // LocationRotateByCurve(doc, element);

                // Rotacionar elemento baseado em LocationPoint
                LocationRotateByPoint(doc, element);

                return Result.Succeeded;
            } catch (Exception ex) {
                TaskDialog.Show("Revit API sample fail: ", ex.Message);
                return ResultFailed;
            }
        }

        /// <summary>
        /// Rotaciona um pilar baseado em seu eixo.
        /// </summary>
        /// <param name="document"></param>
        /// <param name="element"></param>
        public void RotateColumn(Document document, Element element) {
            XYZ point1 = new XYZ(10, 20, 0);
            XYZ point2 = new XYZ(10, 20, 30);
        }
    }
}

```

```

// O eixo deve ser uma bound line
Line axis = Line.CreateBound(point1, point2);

// Usando transacao para girar o pilar
using (Transaction transactionRotateColumn = new Transaction(document,
"Rotacionar pilar."))
{
    transactionRotateColumn.Start();

    // Movemos o pilar antes para o ponto base apenas para melhor
visualiza

    // Mover nao faz parte da rotacao e nao eh necessario
    LocationPoint locationPoint = element.Location as LocationPoint;
    locationPoint.Point = point1;

    // Fazer a rotacao em torno do eixo do elemento
    ElementTransformUtils.RotateElement(document, element.Id, axis,
Math.PI / 3.0);
    transactionRotateColumn.Commit();
}

}

/// <summary>
/// Rotacionar uma parede baseado no LocationCurve.
/// </summary>
/// <param name="document"></param>
/// <param name="element"></param>
/// <returns></returns>
bool LocationRotateByCurve(Document document, Element element) {
    bool rotated = false;

    // Rotacionar o elemento via sua LocationCurve
    using (Transaction transactionLocationRotate = new Transaction(document,
"Rotacionar por LocationCurve."))
    {
        transactionLocationRotate.Start();
        LocationCurve curve = element.Location as LocationCurve;
        if (null != curve) {
            Curve line = curve.Curve;
            XYZ aa = line.GetEndPoint(0);
            XYZ cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
            Line axis = Line.CreateBound(aa, cc);
            rotated = curve.Rotate(axis, Math.PI / 2.0);
        }
        transactionLocationRotate.Commit();
    }
    return rotated;
}

/// <summary>
/// Rotacionar uma parede baseado no LocationPoint.
/// </summary>
/// <param name="document"></param>
/// <param name="element"></param>
/// <returns></returns>
bool LocationRotateByPoint(Document document, Element element) {
    bool rotate = false;

    // Rotacionar o elemento via LocationPoint
    using (Transaction transactionLocationPoint = new Transaction(document,
"Rotacionar por LocationPoint."))
    {
        transactionLocationPoint.Start();

```

```
        LocationPoint location = element.Location as LocationPoint;
        if (null != location) {
            XYZ aa = location.Point;
            XYZ cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
            Line axis = Line.CreateBound(aa, cc);
            rotate = location.Rotate(axis, Math.PI / 2.0);
        }
        transactionLocationPoint.Commit();
    }
    return rotate;
}
}
```

Exemplo - Espelhando Elementos

Iremos analisar agora como proceder para espelhar elementos. Neste exemplo iremos realizar espelhamento em parede. O espelhamento será possível para as classes que possuem a propriedade “Mirrored”.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                // Select an element. In this case, a wall
                Wall wall = doc.GetElement(sel.PickObject(ObjectType.Element, "Select
a wall!")) as Wall;

                // Mirror wall
                MirrorWall(doc, wall);
            }
        }
    }
}
```

```
        return Result.Succeeded;
    } catch (Exception ex) {
        TaskDialog.Show("Revit API sample fail: ", ex.Message);
        return ResultFailed;
    }
}

/// <summary>
/// Mirror wall
/// </summary>
/// <param name="document"></param>
/// <param name="wall"></param>
public void MirrorWall(Document document, Wall wall) {

    Reference reference = HostObjectUtils.GetSideFaces(wall,
    ShellLayerType.Exterior).First();

    // Get one of the wall's major side faces
    Face face = wall.GetGeometryObjectFromReference(reference) as Face;

    UV bboxMin = face.GetBoundingBox().Min;
    // Create a plane based on this side face with an offset of 10 in the x &
    y directions

    Plane plane = Plane.CreateByNormalAndOrigin(face.ComputeNormal(bboxMin),
        face.Evaluate(bboxMin).Add(new XYZ(10, 10, 0)));

    // Using Transaction to mirror
    using (Transaction transactionMirrorWall = new Transaction(document,
    "Mirror wall")) {
        transactionMirrorWall.Start();
        ElementTransformUtils.MirrorElement(document, wall.Id, plane);
        transactionMirrorWall.Commit();
    }
}
}
```

Exemplo - Deletando Elementos e Agrupando Elementos

É possível apagar um elemento em particular ou um grupo de elementos baseado nos IDs dos elementos. Neste exemplo iremos mostrar as duas possibilidades para usar o comando deletar a partir da API do Revit.

Embora seja um tópico não necessariamente relacionado com deletar objetos, neste mesmo exemplo mostramos como agrupar elementos. Criar grupos pode ser útil para manipular um conjunto de elementos como um bloco único.

```
using Autodesk.Revit.ApplicationServices;
```

```

using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                FilteredElementCollector collector = new FilteredElementCollector(doc,
doc.ActiveView.Id);
                IList<Element> allElements = collector.WhereElementIsNotElementType()
                    .WhereElementIsViewIndependent()
                    .ToElements();
                TaskDialog.Show("Revit API", "Total elements: " + allElements.Count);

                // Select an element
                Element oneElement = doc.GetElement(sel.PickObject(ObjectType.Element,
"Select an element!"));

                // Deleting one element
                DeleteElement(doc, oneElement);

                // Delete all the selected elements via the set of element ids
                // Using Transaction to delete elements
                using (Transaction transactionDeleteElements = new Transaction(doc,
"Delete elements")) {
                    transactionDeleteElements.Start();
                    IList<Element> elementsToDelete = sel.PickElementsByRectangle();
                    TaskDialog.Show("Revit API", "Total elements to delete: " +
elementsToDelete.Count);
                    ICollection<ElementId> idSelection = sel.GetElementIds();
                    idSelection.Clear();
                    foreach (Element elem in elementsToDelete) {
                        ElementId id = elem.Id;
                        idSelection.Add(id);
                    }
                    ICollection<ElementId> deletedIdSet = doc.Delete(idSelection);

                    if (0 == deletedIdSet.Count) {
                        throw new Exception("Deleting the selected elements in Revit
failed.");
                    } else {
                        TaskDialog.Show("Revit API", "The selected elements has been
removed.");
                    }
                }
            }
        }
    }
}

```

```

        transactionDeleteElements.Commit();
    }

    // Select elements not deleted
    allElements = collector.WhereElementIsNotElementType()
        .WhereElementIsViewIndependent()
        .ToElements();

    // Grouping elements just to show how to group elements. Not necessary
    to delete
    using (Transaction transactionGroupingElements = new Transaction(doc,
    "Gruping elements")) {
        transactionGroupingElements.Start();

        Group group = null;

        ICollection<ElementId> selectedElementsID = new List<ElementId>();
        foreach (Element element in allElements) {
            selectedElementsID.Add(element.Id);
        }

        if (selectedElementsID.Count > 0) {
            // Group all selected elements
            group = doc.Create.NewGroup(selectedElementsID);
        }

        // Change the default group name to a new name "MyGroup"
        group.GroupType.Name = "MyGroup";

        // Count elements grouped
        int totalElement = 0;
        IList<ElementId> memberIds = group.GetMemberIds();
        foreach (ElementId id in memberIds) {
            totalElement += 1;
        }
        TaskDialog.Show("Revit API", "Total elements grouped: " +
totalElement.ToString());
    }

    transactionGroupingElements.Commit();
}

return Result.Succeeded;

} catch (Exception ex) {
    TaskDialog.Show("Revit API sample fail: ", ex.Message);
    return ResultFailed;
}
}

/// <summary>
/// Delete an element via its ID
/// </summary>
/// <param name="document"></param>
/// <param name="element"></param>
public void DeleteElement(Document document, Element element) {

    ElementId elementId = element.Id;

    // Using Transaction to delete
    using (Transaction transactionDeleteElement = new Transaction(document,
    "Delete element")) {

```

```

        transactionDeleteElement.Start();

        ICollection<ElementId> deleteIdSet = document.Delete(elementId);

        if (0 == deleteIdSet.Count) {
            throw new Exception("Deleting the selected element in Revit
failed.");
        }

        String prompt = "The selected element has been removed and ";
        prompt += deleteIdSet.Count - 1;
        prompt += " more dependent elements have also been removed.";

        // Give the user some information
        TaskDialog.Show("Revit API", prompt);

        transactionDeleteElement.Commit();
    }

}

```

Interagindo com Vistas

O usuário interage com o modelo através de diferentes vistas, que permitem visualizar características especiais. As diferentes visualizações são fundamentais para a interação com o modelo.

Exemplo - Elementos em Vista Ativa

A verificação de elementos em uma determinada vista irá diferir do total no documento inteiro, uma vez que existe elementos não gráficos e mesmo elementos que não estejam sendo visualizados na vista pesquisada. O presente exemplo faz a comparação de paredes visíveis no documento com as presentes na vista ativa.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                CountElements(doc);

                return Result.Succeeded;

            } catch (Exception ex) {
                TaskDialog.Show("Revit API sample fail: ", ex.Message);
                return ResultFailed;
            }
        }

        /// <summary>
        /// Count elements (walls) in the active view
        /// </summary>
        /// <param name="document"></param>
        public void CountElements(Document document) {

            StringBuilder message = new StringBuilder();

            FilteredElementCollector viewCollector = new
FilteredElementCollector(document, document.ActiveView.Id);
            viewCollector.OfCategory(BuiltInCategory.OST_Walls);

            message.AppendLine("Wall category elements within view: " +
viewCollector.ToElementIds().Count);

            FilteredElementCollector docCollector = new
FilteredElementCollector(document);
            docCollector.OfCategory(BuiltInCategory.OST_Walls);
            message.AppendLine("Wall category elements within document: " +
docCollector.ToElementIds().Count);

            TaskDialog.Show("Revit API", message.ToString());
        }
    }
}

```

Exemplo - Tipos de Vista

Cada vista utilizada para visualização do modelo possui uma classificação específica. Pode ser importante definir regras de acordo com a vista em uso, por exemplo, as famílias são criadas para apresentar características específicas para determinado tipo de vista. O exemplo a seguir possibilita determinar o tipo de vista ativo, e verificar o tipo de vista de determinada família de vista.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                View activeView = doc.ActiveView;

                GetViewType(activeView);
                GetViewFamily(doc, activeView);

                return Result.Succeeded;

            } catch (Exception ex) {
                TaskDialog.Show("Revit API sample fail: ", ex.Message);
                return ResultFailed;
            }
        }

        /// <summary>
        /// Determining the view type
        /// </summary>
        /// <param name="view"></param>
        public void GetViewType(View view) {

            // Get the view type of the given view and format the prompt string
            String prompt = "The view is ";

            switch (view.ViewType) {
                case ViewType.Undefined:
                    prompt += "undefined.";
                    break;
                case ViewType.FloorPlan:

```

```
        prompt += "a floor view.";
        break;
    case ViewType.EngineeringPlan:
        prompt += "an engeneering view.";
        break;
    case ViewType.AreaPlan:
        prompt += "an area view.";
        break;
    case ViewType.CeilingPlan:
        prompt += "a ceiling view.";
        break;
    case ViewType.Elevation:
        prompt += "an elevation view.";
        break;
    case ViewType.Section:
        prompt += "a section view.";
        break;
    case ViewType.Detail:
        prompt += "a detail view.";
        break;
    case ViewType.ThreeD:
        prompt += "a threeD view.";
        break;
    case ViewType.Schedule:
        prompt += "a schedule view.";
        break;
    case ViewType.DraftingView:
        prompt += "a drafting view.";
        break;
    case ViewType.DrawingSheet:
        prompt += "a drawing sheet view.";
        break;
    case ViewType.Legend:
        prompt += "a legend view.";
        break;
    case ViewType.Report:
        prompt += "a report view.";
        break;
    case ViewType.ProjectBrowser:
        prompt += "a project browser view.";
        break;
    case ViewType.SystemBrowser:
        prompt += "a system browser view.";
        break;
    case ViewType.CostReport:
        prompt += "a cost report view.";
        break;
    case ViewType.LoadsReport:
        prompt += "a loads report view.";
        break;
    case ViewType.PresureLossReport:
        prompt += "a presure loss view.";
        break;
    case ViewType.PanelSchedule:
        prompt += "a panel schedule view.";
        break;
    case ViewType.ColumnSchedule:
        prompt += "a columns schedule view.";
        break;
    case ViewType.Walkthrough:
        prompt += "a walkthrough view.";
        break;
```

```

        case ViewType.Rendering:
            prompt += "a rendering view.";
            break;
        case ViewType.SystemsAnalysisReport:
            prompt += "a systems analysis report view.";
            break;
        case ViewType.Internal:
            prompt += "an internal view.";
            break;
        default:
            prompt += "undefined.";
            break;
    }
    // Give the user some information
    MessageBox.Show(prompt, "Revit API", MessageBoxButtons.OK);
}

public void GetViewFamily(Document doc, View view) {
    ViewFamily viewFamily = ViewFamily.Invalid;

    ElementId viewType = view.GetTypeId();
    if (viewType.IntegerValue > 1) { // Some views may not have a
        ViewFamilyType viewType = doc.GetElement(viewType) as
        ViewFamilyType;
        viewFamily = viewType.ViewFamily;
    }

    // Give the user some information
    MessageBox.Show(viewFamily.ToString(), "Revit API", MessageBoxButtons.OK);
}

}

```

Exemplo - Criando Vistas Perspectiva e Isométrica 3D

Os exemplos a seguir mostram a criação de vistas. Vistas 3D podem ser criadas com a definição de origem, observador, ponto observado, etc. A seguir um exemplo de criação de uma vista perspectiva.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {

```

```

[Transaction(TransactionMode.Manual)]
class RevitWithWPF : IExternalCommand {
    public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) {

        // Establishing connections
        UIDocument uiDoc = commandData.Application.ActiveUIDocument;
        Document doc = uiDoc.Document;
        Selection sel = uiDoc.Selection;

        try {

            using (Transaction transactionCreatingView = new Transaction(doc,
"Creating Perspective 3D View")) {
                transactionCreatingView.Start();

                // Find a 3D view type
                IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
FilteredElementCollector(doc).OfClass(typeof(ViewFamilyType))
let type = elem as
ViewFamilyType
where
select type;

                type.ViewFamily == ViewFamily.ThreeDimensional

                // Create a new Perspective View3D
                View3D view3D = View3D.CreatePerspective(doc,
viewFamilyTypes.First().Id);
                if (null != view3D) {
                    // By default, the 3D view uses a default orientation
                    // Change the orientation by creating and setting a
ViewOrientation3D
                    XYZ eye = new XYZ(0, -100, 10);
                    XYZ up = new XYZ(0, 0, 1);
                    XYZ forward = new XYZ(0, 1, 0);
                    ViewOrientation3D newViewOrientation3D = new
ViewOrientation3D(eye, up, forward);
                    view3D.SetOrientation(newViewOrientation3D);

                    // Turn off the far clip plane with standard parameter API
                    Parameter farClip =
view3D.get_Parameter(BuiltInParameter.VIEWER_BOUND_ACTIVE_FAR); // BuiltInParameter
Enumeration "Far Clip Active"
                    farClip.Set(0);
                }
                transactionCreatingView.Commit();
            }

            return Result.Succeeded;
        } catch (Exception ex) {
            TaskDialog.Show("Revit API sample fail: ", ex.Message);
            return ResultFailed;
        }
    }
}

```

Utilizando a mesma lógica, é possível criar uma vista isométrica, conforme exemplo a seguir.

```

using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                using (Transaction transactionCreatingView = new Transaction(doc,
                "Creating an Isometric 3D View"))
                transactionCreatingView.Start();

                // Find a 3D view type
                IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
                FilteredElementCollector(doc).OfClass(typeof(ViewFamilyType))
                let type = elem as
                where
                select type;

                ViewFamilyType
                type.ViewFamily == ViewFamily.ThreeDimensional

                // Create a new Isometric View3D
                View3D view3D = View3D.CreateIsometric(doc,
                viewFamilyTypes.First().Id);
                if (null != view3D) {
                    // By default, the 3D view uses a default orientation
                    // Change the orientation by creating and setting a
                    ViewOrientation3D
                    XYZ eye = new XYZ(10, 10, 10);
                    XYZ up = new XYZ(0, 0, 1);
                    XYZ forward = new XYZ(0, 1, 0);
                    ViewOrientation3D newViewOrientation3D = new
                    ViewOrientation3D(eye, up, forward);
                    view3D.SetOrientation(newViewOrientation3D);

                    // Turn off the far clip plane with standard parameter API
                    Parameter farClip =
                    view3D.get_Parameter(BuiltInParameter.VIEWER_BOUND_ACTIVE_FAR); // BuiltInParameter
                    Enumeration "Far Clip Active"
                    farClip.Set(0);

                }
            }
        }
    }
}

```

```
        transactionCreatingView.Commit();
    }

    return Result.Succeeded;

} catch (Exception ex) {
    TaskDialog.Show("Revit API sample fail: ", ex.Message);
    return Result.Failed;
}
}

}
```

Interagindo com Caixas de Corte

As vistas 3D possuem uma caixa de corte, Section Box, para limitar a exibição dos elementos ao conteúdo limitado por essa caixa. Estas caixas são bem interessantes para visualização parcial de elementos e para ver elementos que seriam obstruídos por outros que estão entre o ponto do visualizador e o objeto a ser visualizado.

Exemplo - Caixa de Corte

O exemplo a seguir mostra três funcionalidade relacionadas à caixa de corte. O primeiro mostra como desativar/ativar a caixa de corte ativa na vista, após definir uma vista 3D, tal como feito em exemplos anteriores. A segunda funcionalidade mostra como expandir uma caixa de corte, igualmente após ter sido criada uma vista 3D. Um terceiro aprendizado é obter as coordenadas da caixa de corte em WCS, sistema de coordenadas global.

```
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
```

```

namespace RevitCommand {
    [Transaction(TransactionMode.Manual)]
    class RevitWithWPF : IExternalCommand {
        public Result Execute(ExternalCommandData commandData, ref string message,
        ElementSet elements) {

            // Establishing connections
            UIDocument uiDoc = commandData.Application.ActiveUIDocument;
            Document doc = uiDoc.Document;
            Selection sel = uiDoc.Selection;

            try {

                using (Transaction transaction3DViewsSectionBox = new Transaction(doc,
                "Working with 3D Views Section Box")) {
                    transaction3DViewsSectionBox.Start();

                    // Find a 3D view type
                    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new
                    FilteredElementCollector(doc).OfClass(typeof(ViewFamilyType))
                    ViewFamilyType
                    type.ViewFamily == ViewFamily.ThreeDimensional
                    let type = elem as
                    where
                    select type;

                    // Create a new Isometric View3D
                    View3D view3D = View3D.CreateIsometric(doc,
                    viewFamilyTypes.First().Id);

                    // Showing the Section Box
                    ShowHideSection(view3D);

                    // Expanding and Hiding the Section Box
                    ExpandSectionBox(view3D);

                    transaction3DViewsSectionBox.Commit();
                }

                return Result.Succeeded;
            } catch (Exception ex) {
                TaskDialog.Show("Revit API sample fail!", ex.Message);
                return ResultFailed;
            }
        }
        /// <summary>
        /// Showing the Section Box
        /// </summary>
        private void ShowHideSection(View3D view3D) {
            foreach (Parameter p in view3D.Parameters) {
                // Get Section Box parameter
                if (p.Definition.Name.Equals("Section Box")) {
                    // Show Section Box
                    p.Set(1);
                    // Hide Section Box
                    // p.Set(0);
                    break;
                }
            }
        }
    }
}

```

```
/// <summary>
/// Expanding and Hiding the Section Box
/// </summary>
/// <param name="view"></param>
private void ExpandSectionBox(View3D view) {
    // The original section box
    BoundingBoxXYZ sectionBox = view.GetSectionBox();

    // Expand the section box
    XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
    sectionBox.Max += deltaXYZ / 2;
    sectionBox.Min -= deltaXYZ / 2;

    // After resetting the section box, it will be show in the view
    // It only works when the Section Box check box is checked in
    // View property dialog
    view.SetSectionBox(sectionBox);

    // To deactivate the section box
    // view.IsSectionBoxActive = false;

}

private void ConvertMaxMintowCS(View3D view, out XYZ max, out XYZ min) {
    BoundingBoxXYZ sectionbox = view.GetSectionBox();
    Transform transform = sectionbox.Transform;
    max = transform.OfPoint(sectionbox.Max);
    min = transform.OfPoint(sectionbox.Min);
}

}
```

eTlipse

Referências

Dynamo Unchained 1: Learn how to develop Zero Touch Nodes in C# - [http://teocomi.com/dynamo-unchained-1-learn-how-to-developzero-touch-nodes-in-csharp/](http://teocomi.com/dynamo-unchained-1-learn-how-to-develop-zero-touch-nodes-in-csharp/)

Dynamo Unchained 2: Learn how to develop explicit Custom Nodes in C# -
<http://teocomi.com/dynamo-unchained-2-learn-how-to-develop-explicit-nodes-in-csharp/>

Revit King

<http://revitking.blogspot.com/2012/01/want-to-learn-revit-api-from-building.html>

Revit API Docs

<https://www.revitapidocs.com/>

The Building Coder – Jeremy Tammik Jeremy Tammik – Programming Forge, BIM and the Revit API

<https://thebuildingcoder.typepad.com/>

Archi-lab Revit API

<https://archi-lab.net/tag/revit-api/>

Revit API Developer Guide

https://help.autodesk.com/view/RVT/2019/ENU/?guid=Revit_API_Revit_API_Developers_Guide.html

Harry Mattison's Revit API Blog

<https://boostyourbim.wordpress.com/>

Danny Bentley's Revit API Youtube Channel

<https://www.youtube.com/watch?v=C0mNU2bEUSs&list=PLlyMZ5lcKcci1TvB4qM9S8J-RKp0DhVWO>

Revit Lookup Installer

<https://boostyourbim.wordpress.com/2018/05/17/revit-lookup-2019-installer/>

Pushing Revit to the Next Level: An Intro to Revit Plugins with C# – Jeremy Graham

<https://www.autodesk.com/autodesk-university/class/Pushing-Revit-Next-Level-Intro-Revit-Plugins-C-2018>

Treinamento Profissional em C# .net – Herbert Moroni

Dynamo Language Manual

https://dynamobim.org/wp-content/uploads/forum-assets/colin-mccroneautodesk-com/07/10/Dynamo_language_guide_version_1.pdf