[SD468797-L]

The Cross-Platform Revit: Sharing Code with Plug-Ins, Dynamo, and Forge

Thiago Almeida
Blackbird Industries

Eduardo Thiesen
Onbox

Learning Objectives

- Create structured and modular code using the inversion of control principle.
- Share data specific code between different programming languages
- Share nonspecific code between different environments like desktop, cloud and even front-end web
- Share Revit-specific code between cross-Revit environments like addi-ins, Dynamo, and Design Automation

Description

The software industry has changed quite a lot recently; Autodesk is heavily investing in porting their most important packages to be accessible in the cloud. Revit software, for instance, can run as a part of the Forge platform. Microsoft is investing tremendously in its Dotnet platform development (which was rewritten from scratch to be open source and modular). When dealing with Revit API development, however, we often see monolithic apps that directly couple modules and components like the UI layer to gather input, introducing several technical challenges. Trying to manage and scale this type of codebase is frustrating and time consuming, but it doesn't have to be. In this lab, you will discover a free and open-source framework that helps you quickly prototype Revit applications, while providing you with an easy way to develop modular code that can be portable to multiforms of Revit environments, like add-ins, Dynamo, and even the Forge Design Automation API.

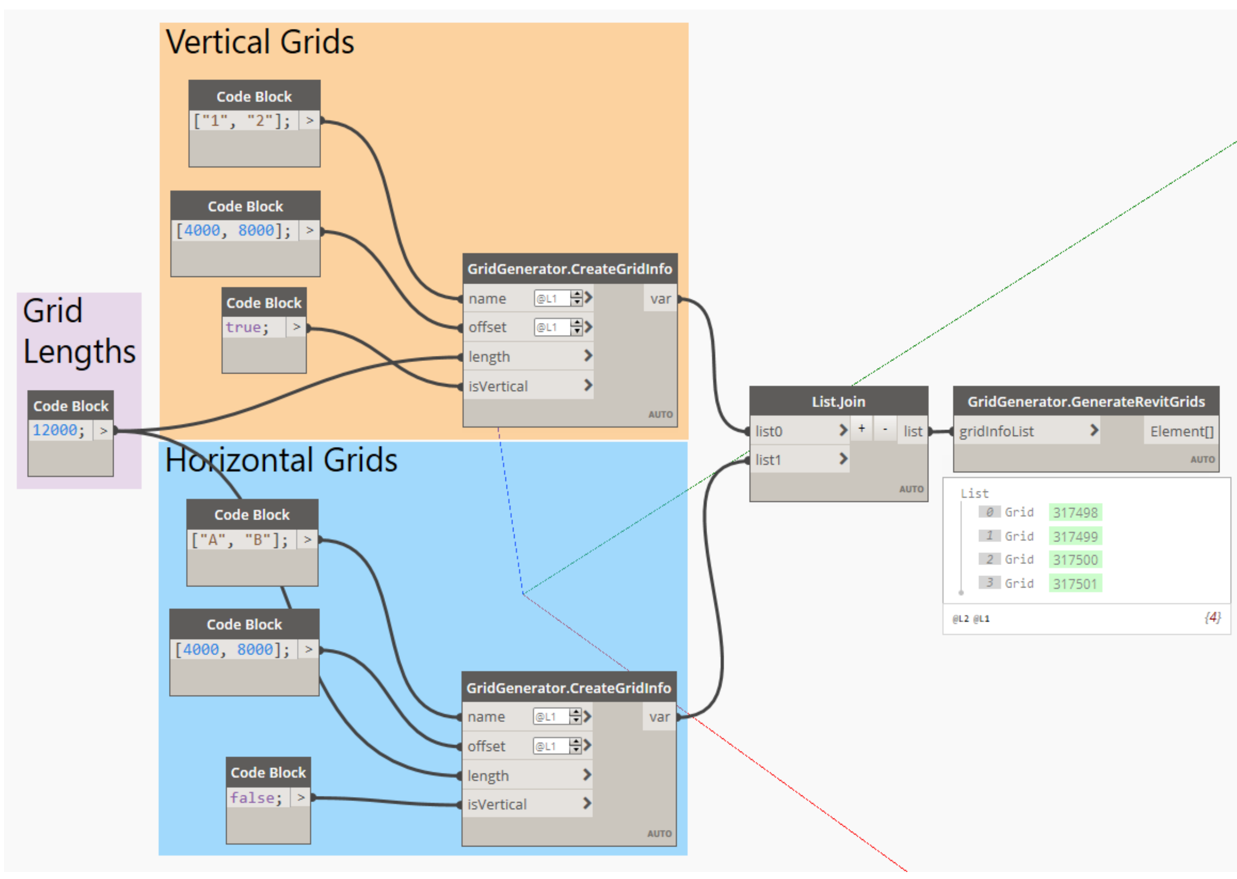
Speaker(s)

Thiago Almeida has a Bachelor Degree in Civil Engineering and in Product Design. Technician Degree in Software Analysis. Strong knowledge of engineering, mathematics and Autodesk products including official Autodesk certifications on Revit Architecture, Revit Structure, Revit MEP and 3d Studio Max. For the past 7 years started to dive into the Revit API to explore opportunities and to leverage customizable Building Information Modeling workflows for clients and partners, resulting in 6 products running in production to thousands of active users.

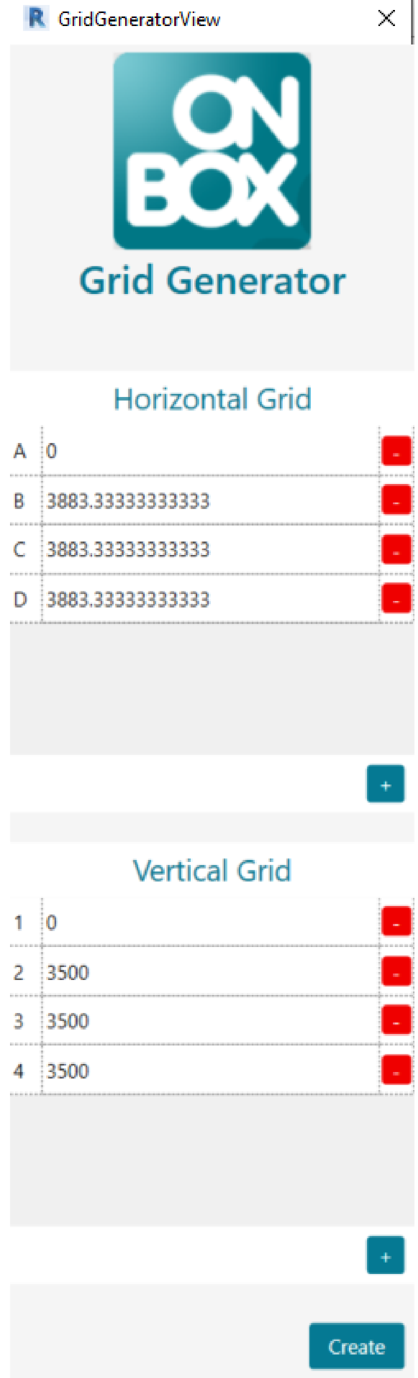
Eduardo Thiesen has recently acquired his bachelor's degree in civil engineering. Started working with the Revit API in 2018 and has been working with Design Automation for Revit since it's beta release. He is responsible for the integration between Shedmate and Revit, allowing for material takeoffs, fabrication drawings and ready for use drawing detailing to be generated from scratch on the cloud.

In this lab you will go through the entire process of creating a Revit application, a Dynamo package and a Design automation package using a modular and scalable approach. The following images show all different implementations working.

Dynamo



Revit Application



Angular web page hooked to Design Automation

ON BOX

Grid Generator

Horizontal Grids

A	0
B	2000
C	2000

Vertical Grids

1	0
2	2000
3	2000

1 2 3

A

B

C

Introducing the Onbox framework

Onbox is a free and open source framework to help you build modern cross platform Revit applications in a similar fashion of Angular and ASP.Net core.

The full documentation can be found at https://engthiago.github.io/Onboxframework.docs\

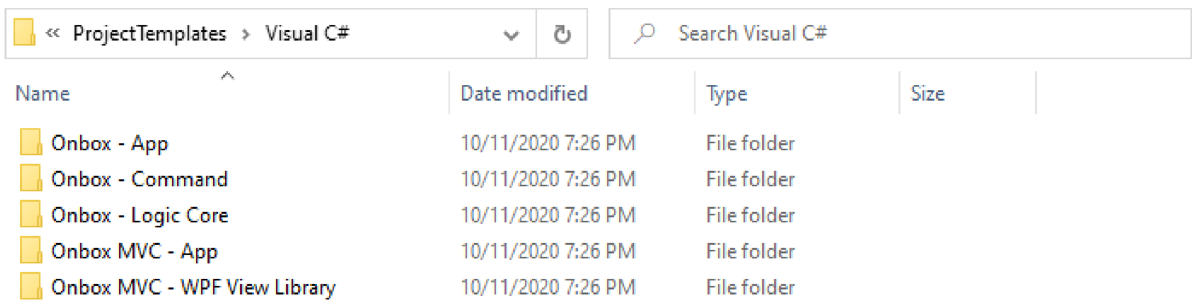
Getting Started – Creating a monolithic Revit application

We will start by creating a simple Revit application using the framework and move on to a more modular approach later.

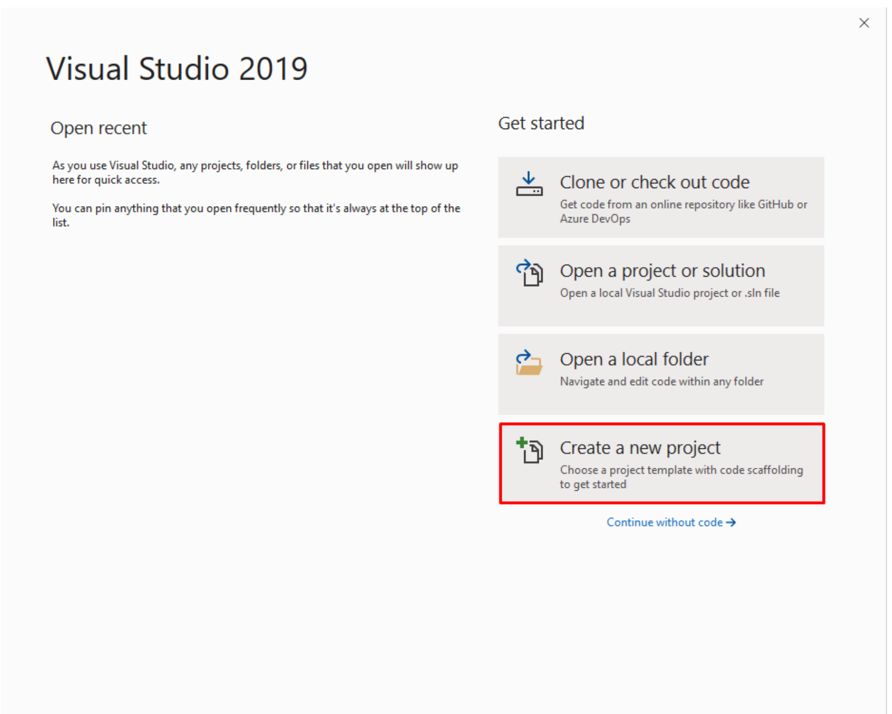
1. Download and Install Visual Studio Templates.

Download the template files at https://github.com/engthiago/Onboxframework.docs/releases. Then unzip the contents of the folder in *%userprofile%\Documents\Visual Studio 20**\Templates* folder.

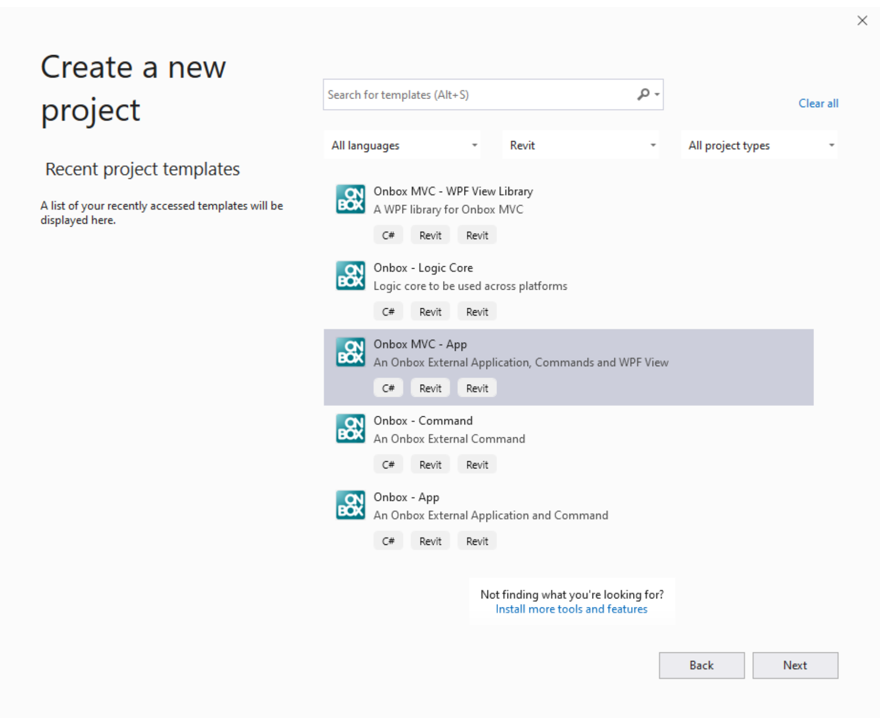
2. The resulting folder structure will be something like this for the "ProjectTemplates" folder:



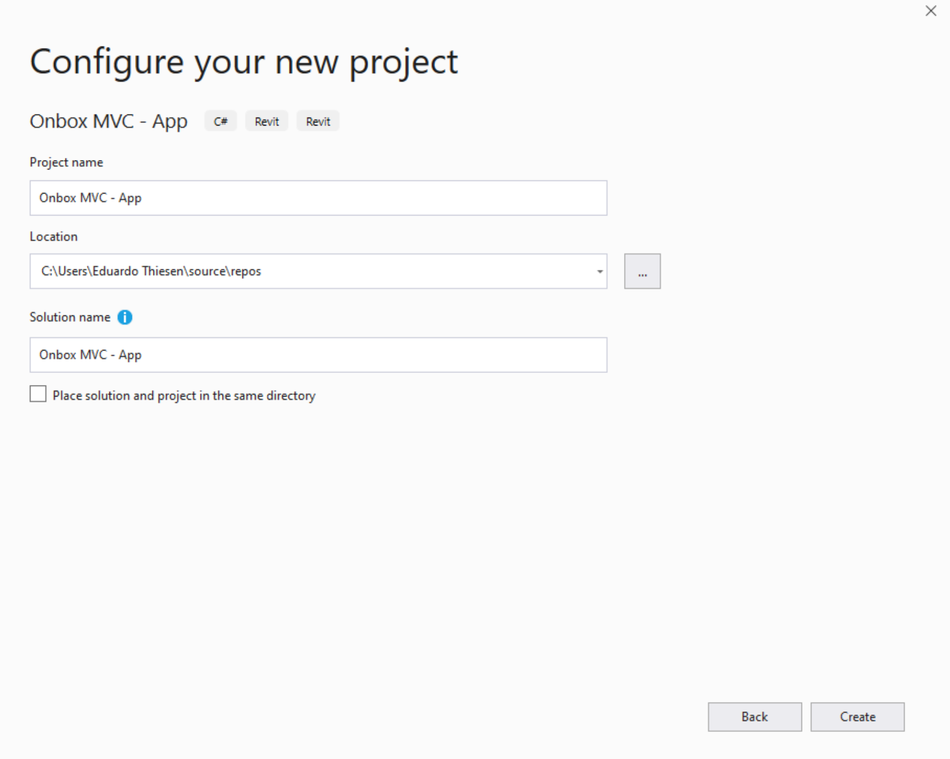
3. Launch Visual Studio and click on “Create a new project”.



4. On the Platform drop down menu, choose Revit and then pick “Onbox MVC – APP".

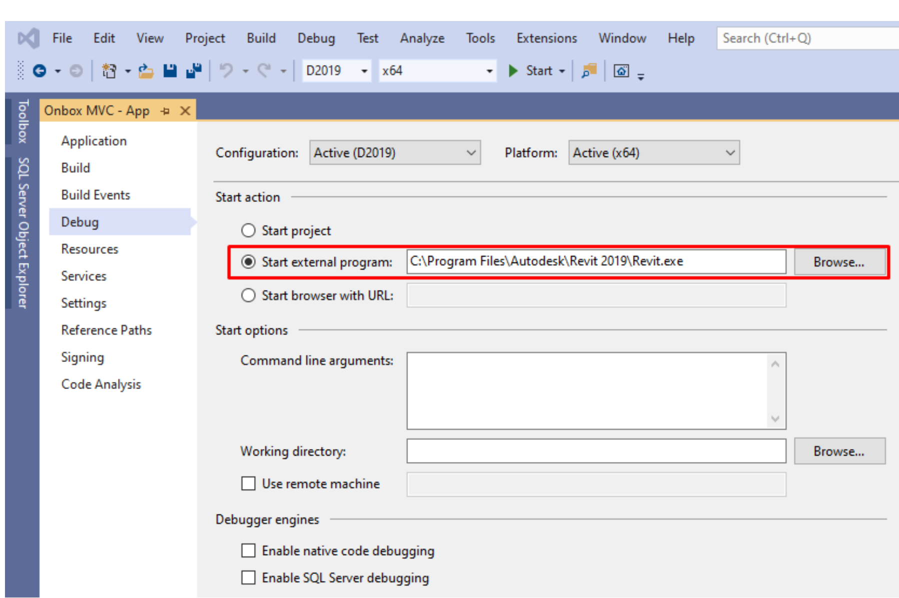


5. Type your project name and click “Create”.

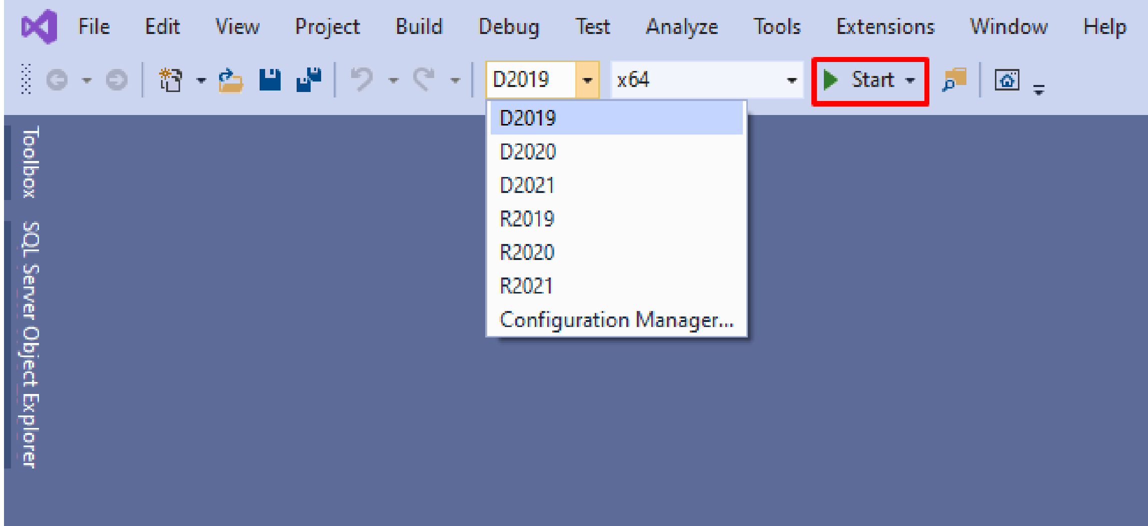


Launch your application

6. Check your Revit installation folder, if it is on the default *C:\Program Files\Autodesk\Revit* folder, move on to the next step. Otherwise, go to “Project Properties => Debug” and change the path to your Revit install location.

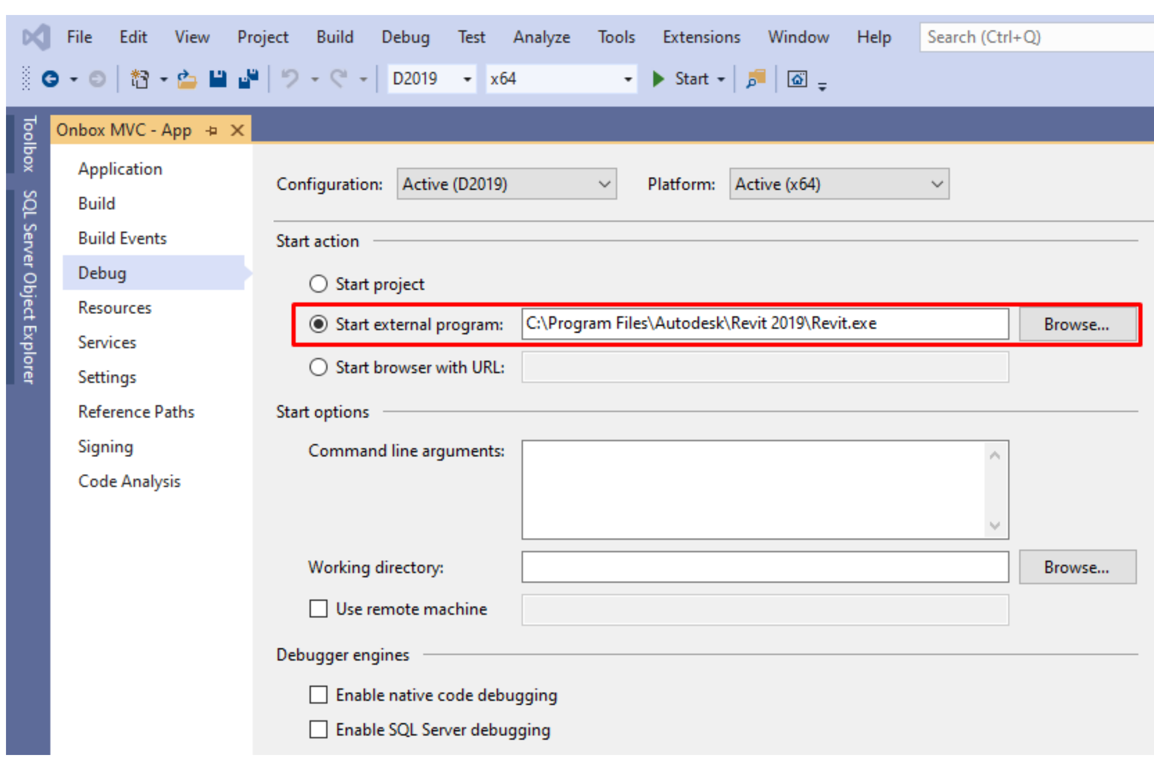


7. Choose a Revit version on the Solution Configuration Drop Down menu and hit “Start”.

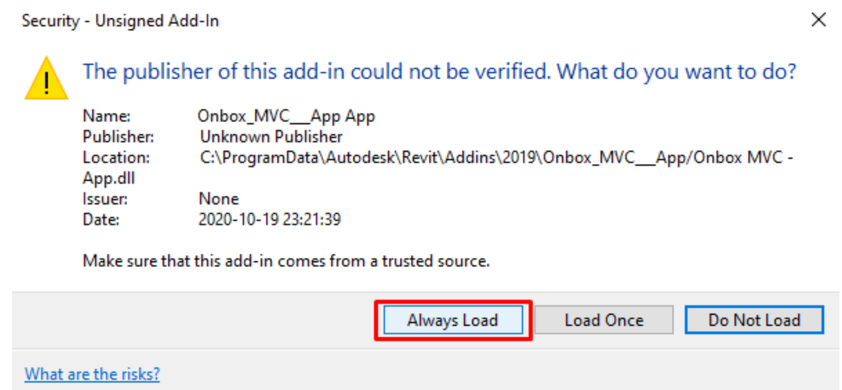


First build troubleshooting

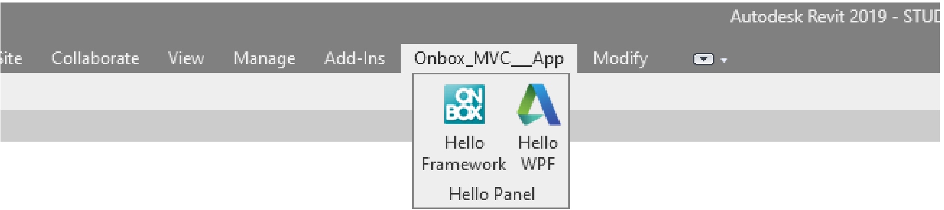
Visual Studio might complain when you first try to build your application, this happens because of a bug on PropertyChanged.Fody when referenced by NuGet. If you get build errors in the previous subsection, go to “Build => Clean Solution” and try to run the solution again. Once the packages are downloaded you should not have this issue anymore.



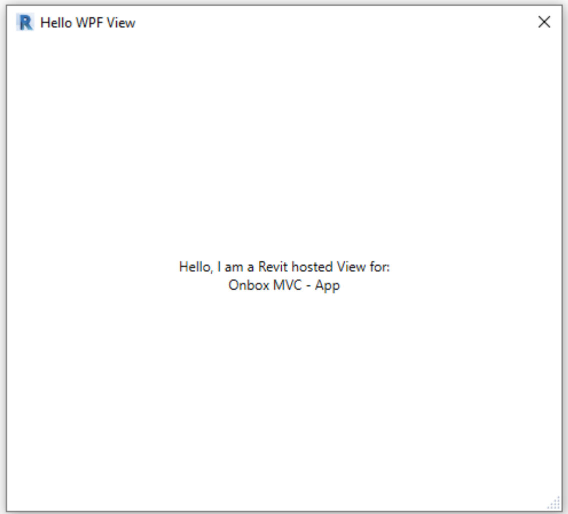
8. Accept the loading of the Addin.



9. After launching Revit, open or create a project and you will see a new Ribbon tab with two buttons:



10. Clicking in the button “Hello WPF” will show this dialog:



Congratulations! You have your application up and running.

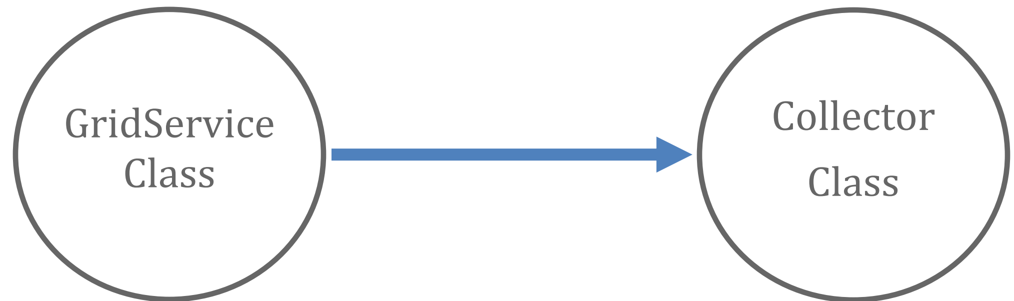
Dependency Injection

Dependency Injection is a technique based on the inversion of control principle, where one object supplies the dependencies of another object. It allows us to abstract the creation of types from the places where these types will be consumed, that way you can modify what type will be created and how it will be created, without requiring you to refactor most of your code. With that we can write cleaner and more loosely coupled code.

Please note that explaining Dependency Injection in depth is outside the scope of this presentation, so we will only go through some of its concepts and how they are applied in the framework.

Class Dependencies

Let's say we have a class that will perform operations on grids, we'll name it GridService, and that this class will have a method responsible for deleting all grids within the project. Before the grids can be deleted we need to collect them, for that we will use another class, that we'll call Collector. So we can say that GridService depends on the Collector class.



Here is what it would look like in code:

```
public class GridService
{
    public ICollection<ElementId> DeleteGrids(Document doc)
    {
        var collector = new Collector();
        var grids = collector.CollectAll<Grid>(doc);

        return doc.Delete(grids);
    }
}
```

Class Decoupling

Notice that we need to instantiate the collector class inside the DeleteGrids method. Let's say we need to add another argument to the Collector constructor, you could just edit the class and change the instatiation manually. Now, imagine this class is being used in dozens of methods and classes, this would become a maintenance nightmare really quickly.

We solve that by applying class decoupling. The first step is to move the creation of the collector to outside the GridService, in this case we will have a field in the GridService class that will hold an instance of the Collector. Then we create a constructor to take the Collector as an argument and store it locally. Like so:

```
public class GridService
{
    private readonly ICollector collector;

    public GridService(ICollector collector)
    {
        this.collector = collector;
    }

    public ICollection<ElementId> DeleteGrids(Document doc)
    {
        var grids = this.collector.CollectAll<Grid>(doc);
        return doc.Delete(grids);
    }
}
```

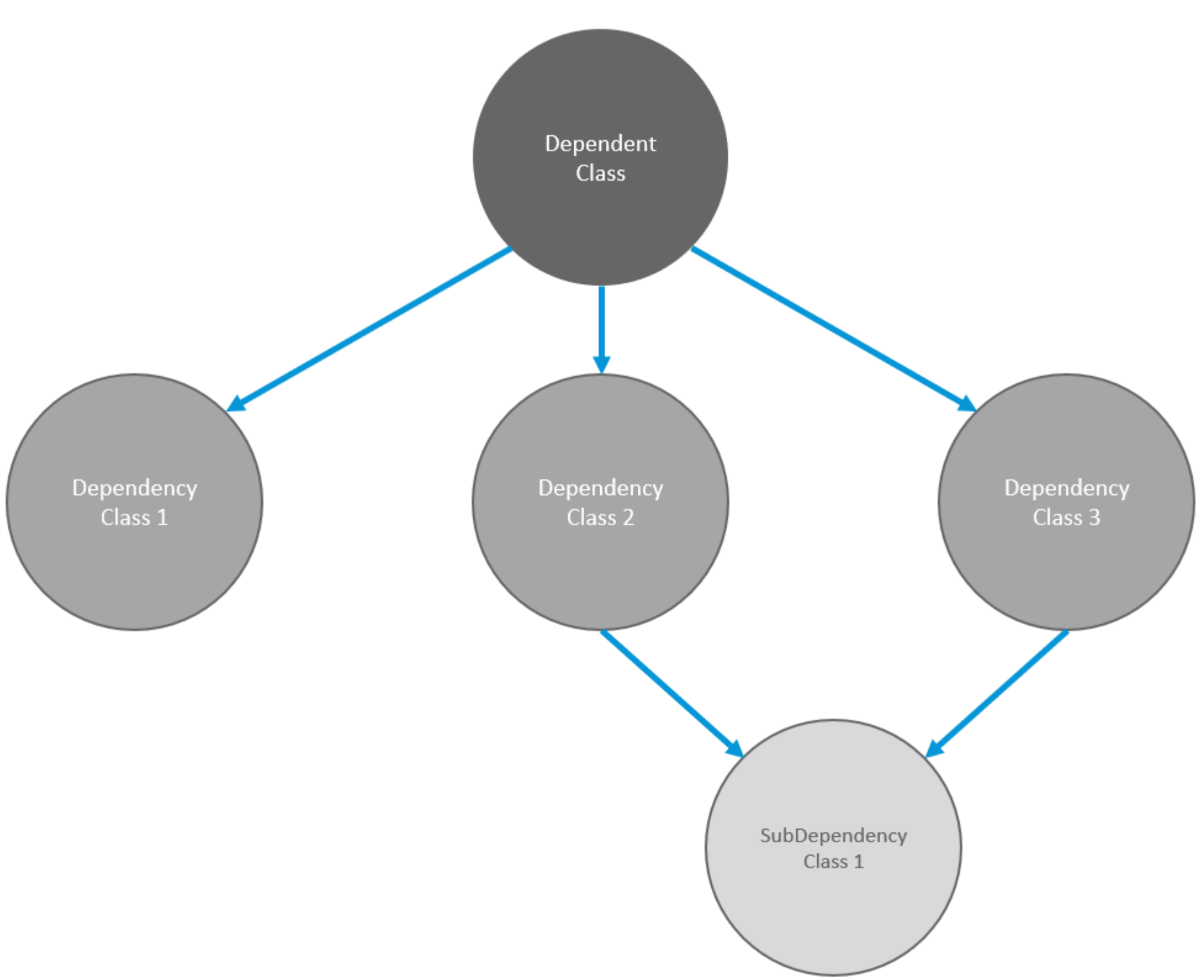
In this example the control of the class has been inverted, because it has no responsibility to instantiate its dependency, the Collector. We have also gone one step further, by swapping the Collector type with an abstraction, that way we can use different types of Collectors and pass it in the GridService if we need to, without having to go change anything inside it.

Next, we'll take a look at how these classes will be consumed:

```
var collector = new Collector();
var gridService = new GridService(collector);
gridService.DeleteGrids(doc);
```

Class Hirearchy

The previous example may seem good enough, but the truth is the GridService is a very simple class, let's take a look at a more complex situation, more similar to real world scenarios.



In order to instantiate de DependentClass we would need the following code:

```
var dependencyClass1 = new DependencyClass1();
var subDependencyClass1 = new SubDependencyClass1();
var dependencyClass2 = new DependencyClass2(subDependencyClass1);
var dependencyClass3 = new DependencyClass3(subDependencyClass1);

var dependentClass = new DependentClass
(
    dependencyClass1,
    dependencyClass2,
    dependencyClass3
);
```

So in this situation the inversion of control just moved the problem, instead of solving it, since for every DependentClass instance we would need to deal with all those constructors again. So, how can we actually solve the problem?

IOC Container

One way to approach this issue is through the use of Inversion of Control containers. That way we can ask the container to resolve the dependencies of a type, like this:

```
var dependentClass = container.Resolve<DependentClass>();
```

This single line of code will return an instance of the class, that could be a new instance or a cached one, depending on the context of that container. But how does the container now how to instantiate the classes, and whether it should be cached or not?

Containers have three main lifecycle phases: Register, where you tell the container how the classes should be instantiated; Resolve, where it creates the instances, like in the example above; and Dispose, which is self explanatory. Now that may seem very complex but the Onbox Framework simplifies that process a lot, and that will be shown in the next section.

Project Structure

Traditional Revit Application

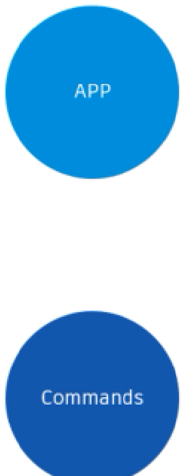
Now let's talk about a traditional Revit application. It has the application itself, which is responsible for creating the ribbon UI and initialize any application logic, and is often used to hold variables that need to be kept throughout its lifecycle. Then, it has a set of commands that will execute some logic when the user interacts with them. The following is an example of a Revit application class.

```
public class RegularRevitApp : IExternalApplication
{
    public Result OnStartup(UIControlledApplication application)
    {
        // Create Ribbon
        // Start up any application logic

        return Result.Succeeded;
    }

    // Omitted rest of the class
}
```

The issue with that approach is that any communication between the application and the command is done in a manual way, you can store variables and tell the command to access them in Revit, but there is no association between them.



Now, let’s talk about the application inside the framework. It will, by design, have an uniquely identified container attached to it, this is the container that the application owns and is the one responsible for initializing and registering dependencies on it. Then it will have a set of commands that will have a scoped version of the container attached to it, which is essentially a copy of the application’s container.

The containers will be linked through a command class that references the application as its parent. Let’s see how that works in code:

```
[ContainerProvider("5cc7afd4-bedf-4396-beaf-47ce11d70f5c")]
public class App : RevitApp
{
    public override Result OnStartup(IContainer container, UIControlledApplication application)
    {
        // Add Dependencies to the container
        // Start up any application logic

        return Result.Succeeded;
    }

    // Omitted rest of the class
}
```

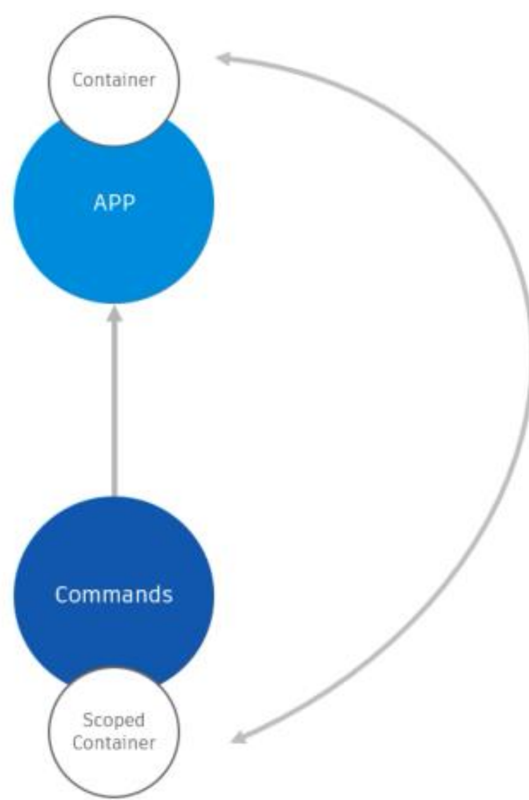
Note that we are inheriting from the RevitApp class, instead of the IExternalApplication interface, and we have a ContainerProvider decorator, this is how we attach the container to the specific application. Then, on the Startup method we can have a container injected for us, and can start registering the dependencies that we need.

Now let’s take a look at the command class implementation. Again, you’ll see that we inherit from the RevitAppCommand class and not IExternalCommand, and we have the injection of a ContainerResolver on the executed method, these are the only differences we have from a regular Revit command.

```csharp
[Transaction(TransactionMode.Manual)]
public class Command : RevitAppCommand<App>
{
    public override Result Execute(
                        IContainerResolver container,
                        ExternalCommandData commandData,
                        ref string message,
                        ElementSet elements)
    {
        // Resolve any necessary dependency
        // Execute logic

        return Result.Succeeded;
    }
}
```

A container resolver is designed to resolve any dependencies that where registered in our container.



Creating the Grid Generator

Here is a summary of the classes contained in our GridGenerator app. To see the entire code, head to the github links below.

- https://github.com/engthiago/Onbox.GridGenerator.Monolithic - Monolithic version
- https://github.com/engthiago/Onbox.GridGenerator.Modular - Modular version
- https://engthiago.github.io/Onboxframework.docs/tutorials/1_guetstart.html - Framework documentation.
- https://github.com/engthiago/Onboxframework.docs/releases - To download the Framework's Visual Studio templates.

Revit application

1. App.cs

This is the entry point of your application, here you will define the lifecycle hooks for creating the app and integrating with Revit.

```
[ContainerProvider("8fe7986c-9cea-49ad-be3c-a2ca57a1bb2a")]
1 reference
public class App : RevitApp
{
    0 references
    public override void OnCreateRibbon(IRibbonManager ribbonManager)
    {
        // Here you can create Ribbon tabs, panels and buttons
        var br = ribbonManager.GetLineBreak();

        // Adds a Ribbon Panel to the Addins tab
        var addinPanelManager = ribbonManager.CreatePanel("Grid Generator");
        addinPanelManager.AddPushButton<GridGeneratorCommand, AvailableOnProject>($"Grid{br}Generator", "onbox_logo");
    }
    0 references
    public override Result OnStartup(IContainer container, UIControlledApplication application)
    {
        // Here you can add all necessary dependencies to the container
        container.AddOnboxCore();
        container.AddRevitMvc();

        container.AddUnitConverters();

        container.AddGridGenerator();
        container.AddViews();

        return Result.Succeeded;
    }
    0 references
    public override Result OnShutdown(IContainerResolver container, UIControlledApplication application)...
}
```

Models

2. GridInfo.cs

Represents grid information that will be displayed on the UI and used to calculate the position of the Revit grids.

```
public class GridInfo
{
    7 references
    public bool IsVertical { get; set; }
    4 references
    public string Name { get; set; }
    5 references
    public double Offset { get; set; }
    4 references
    public double TotalOffset { get; set; }
    3 references
    public double Length { get; set; }
}
```

3. GridSettings.cs

Contains the default settings for the grids that will be generated.

```
public class GridSettings
{
    3 references
    public double DefaultOffset { get; set; }
    1 reference
    public double MinimumOffset { get; set; }
    5 references
    public double DefaultLength { get; set; }
    3 references
    public double MinimumLength { get; set; }
}
```

Services

Services are classes that should only contain methods and hold no state, properties or fields. It will never be directly instantiated, only injected by the container. Also, its methods should never receive other services, only data models, simple parameters or external references.

4. GridCollectorService.cs

Collects existing grids on the project.

```
public class GridCollector : IGridCollector
{
    0 references
    public GridCollector()...

    2 references
    public ICollection<ElementId> CollectAllGrids(Document doc)...

    2 references
    public List<Grid> CollectHorizontalGrids(Document doc)...

    2 references
    public List<Grid> CollectVerticalGrids(Document doc)...
}
```

5. RevitLengthConverter.cs

Holds all length dimension conversion methods.

```
public class RevitLengthConverter : ILengthConverter
{
    5 references
    public double ConvertInternalToMillimeters(double feet)...

    3 references
    public double ConvertMillimetersToInternal(double millimiters)...
}
```

6. GridInfoFactory.cs

Responsible for creating instances of the GridInfo class.

```
public class GridInfoFactory : IGridInfoFactory
{
    private readonly GridSettings gridSettings;

    0 references
    public GridInfoFactory(GridSettings gridSettings)...

    2 references
    public GridInfo CreateVertical(string name,
                                   double previousOffset)...

    2 references
    public GridInfo CreateVertical(string name,
                                   double offset,
                                   double previousOffset)...

    2 references
    public GridInfo CreateVertical(string name,
                                   double offset,
                                   double previousOffset,
                                   double length)...

    2 references
    public GridInfo CreateHorizontal(string name,
                                     double previousOffset)...

    2 references
    public GridInfo CreateHorizontal(string name,
                                     double offset,
                                     double previousOffset)...

    2 references
    public GridInfo CreateHorizontal(string name,
                                     double offset,
                                     double previousOffset,
                                     double length)...

    6 references
    private GridInfo Create(string name,
                            double offset,
                            double previousOffset,
                            double length)...
}
```

7. GridInfoService.cs

Calculates the parameters for each GridInfo.

```
public class GridInfoService : IGridInfoService
{
    private readonly GridSettings gridSettings;
    private readonly IGridInfoFactory gridInfoFactory;

    0 references
    public GridInfoService(
        GridSettings gridSettings,
        IGridInfoFactory gridInfoFactory
        )...

    3 references
    public string GetGridName(int gridIndex, bool isVertical)...

    2 references
    public double GetTotalOffset(List<GridInfo> grids)...

    2 references
    public double GetGridTotalOffset(GridInfo grid, List<GridInfo> grids)...

    2 references
    public double GetGridLength(List<GridInfo> perpendicularGrids)...

    3 references
    public List<GridInfo> AddNewGrid(List<GridInfo> parallelGrids,
                                     bool isVertical)...

    3 references
    public List<GridInfo> RemoveGrid(List<GridInfo> grids,
                                     GridInfo gridToRemove,
                                     bool isVertical)...

    3 references
    public List<GridInfo> UpdateGrids(List<GridInfo> parallelGrids,
                                      List<GridInfo> perpendicularGrids,
                                      bool isVertical)...


}
```

8. RevitGridService.cs

Converts information from Revit grids to GridInfo and manipulates existing Revit grids.

```
public class RevitGridService : IRevitGridService
{
    private readonly IGridInfoFactory gridInfoFactory;
    private readonly IGridCollector gridCollector;
    private readonly ILengthConverter lengthConverter;

    0 references
    public RevitGridService(
        IGridInfoFactory gridInfoFactory,
        IGridCollector gridCollector,
        ILengthConverter lengthConverter
        )...

    2 references
    public ICollection<ElementId> DeleteGrids(Document doc)...

    2 references
    public List<Grid> CreateRevitGrids(Document doc, List<GridInfo> gridInfoList)...

    4 references
    private Line GetGridLine(Grid grid)...

    1 reference
    public double GetGridXOffset(Grid grid)...

    1 reference
    public double GetGridYOffset(Grid grid)...

    3 references
    public List<GridInfo> ConvertToGridInfoList(List<Grid> grids, bool areVertical)...

    1 reference
    private GridInfo ConvertToHorizontalGridInfo(Grid grid, double currentOffset)...

    1 reference
    private GridInfo ConvertToVerticalGridInfo(Grid grid, double currentOffset)...
}
```

Views

The main Onbox.MVC library is designed with Revit in mind but it can actually be used on any regular Windows WPF application as it has no refence to Revit on its code. In the other hand Onbox.MVC.Revit provides functionality specific Revit.

9. GridGeneratorView.xaml

Contains the UI for the view.

10. GridGeneratorView.xaml.cs

Code behind for the WPF view, ideally methods in this class should be events that will hook into services for their functionality.

Resources

Contains all static resources of the project. Note that ribbon images need to contain two versions with different dimensions, one with 16x16 pixels and one with 32x32, and follow the naming convention found in the template examples.

Commands

11. GridGeneratorCommand.cs

```
[Transaction(TransactionMode.Manual)]
1 reference
public class GridGeneratorCommand : RevitAppCommand<App>
{
    0 references
    public override Result Execute(IContainerResolver container,
                                   ExternalCommandData commandData,
                                   ref string message,
                                   ElementSet elements)
    {
        var doc = commandData.Application.ActiveUIDocument.Document;

        // Get current grids
        var gridCollector = container.Resolve<IGridCollector>();
        var horizontalGrids = gridCollector.CollectHorizontalGrids(doc);
        var verticalGrids = gridCollector.CollectVerticalGrids(doc);

        // Convert Grids to GridInfo
        var revitGridService = container.Resolve<IRevitGridService>();
        var horizontalGridInfoList = revitGridService.ConvertToGridInfoList(horizontalGrids, false);
        var verticalGridInfoList = revitGridService.ConvertToGridInfoList(verticalGrids, true);

        // Instantiate view and set grid info lists
        var gridGeneratorView = container.Resolve<IGridGeneratorView>();
        gridGeneratorView.SetHorizontalGrids(horizontalGridInfoList);
        gridGeneratorView.SetVerticalGrids(verticalGridInfoList);

        var dialogResult = gridGeneratorView.ShowDialog();
        if (dialogResult == true)
        {
            var grids = new List<GridInfo>();
            grids.AddRange(gridGeneratorView.GetHorizontalGrids());
            grids.AddRange(gridGeneratorView.GetVerticalGrids());

            using (var t = new Transaction(doc, "Create grids"))
            {
                t.Start();

                // Clear Existing grids and create the new ones
                revitGridService.DeleteGrids(doc);
                revitGridService.CreateRevitGrids(doc, grids);

                t.Commit();
            }
        }

        return Result.Succeeded;
    }
}
```

Container Extensions

In order to make the code more modular you can create different container extensions that will inject a specific part of the application logic.

12. GridGeneratorExtensions.cs

Registers all classes responsible for the grid generator, and the default grid settings to the container.

```
public static class GridGeneratorExtensions
{
    1 reference
    public static IContainer AddGridGenerator(this IContainer container)
    {
        var gridSettings = new GridSettings
        {
            DefaultLength = 4000,
            MinimumLength = 500,
            DefaultOffset = 2000,
            MinimumOffset = 500
        };
        container.AddSingleton(gridSettings);

        container.AddTransient<IGridInfoFactory, GridInfoFactory>();
        container.AddTransient<IGridInfoService, GridInfoService>();
        container.AddTransient<IGridGeneratorView, GridGeneratorView>();
        container.AddTransient<IRevitGridService, RevitGridService>();
        container.AddTransient<IGridCollector, GridCollector>();

        return container;
    }
}
```

13. UnitConverterExtensions.cs

Registers the RevitLengthConverter class to the container.

```
public static class UnitConverterExtensions
{
    1 reference
    public static IContainer AddUnitConverters(this IContainer container)
    {
        container.AddSingleton<ILengthConverter, RevitLengthConverter>();

        return container;
    }
}
```

14. ViewsExtensions.cs

Registers the MessageBoxService to the container. This service is included in the Onbox NuGet packages.

```
public static class ViewsExtensions
{
    1 reference
    public static IContainer AddViews(this IContainer container)
    {
        // Adds MessageBoxService to the container
        container.AddSingleton<IMessageService, MessageBoxService>();

        return container;
    }
}
```

Abstractions

The abstractions folder contains interfaces for the view and all services on the project. This is very important for modularity and testability, since you can choose to inject different implementations of the same interface depending on the context.