Calculator WPF Application Document

The calculator desktop application is developed with the following technologies:

- ❖ WPF application
- ❖ MVVM pattern
- ❖ Caliburn.Micro framework is used
- ❖ IoC Container is implemented
- ❖ Application has one dialog window
- ❖ Serilog Logger is implemented

Conceptualization

This section outlines specific concepts for understanding the technologies used in this application. Finally, a detailed description of how these techniques are implemented.

MVVM (Model-View-ViewModel)

The Model View ViewModel (MVVM) is an architectural pattern, and MVVM is a way to create client applications that leverage core features of the WPF platform, enable easy unit testing of application functionality, and help developers and designers collaborate on minor technical difficulties.

VIEW: A view is defined in XAML and should have no logic in the code behind it. It binds to the view model using only the data binding. The view contains the visual controls that are displayed to the user and may also contain animations, navigation aspects, themes, and other interactive features for the purpose of visual presentation.

MODEL: The model is responsible for providing data in such a way that it can be easily processed by WPF. It may need to implement INotifyPropertyChanged and/or INotifyCollectionChanged. If retrieving data is expensive, it abstracts the expensive operations and never blocks the UI thread. It is the data or business logic that is completely independent of the UI that stores state and performs problem domain processing. The model is written in code or represented by pure data coded in relational tables or XML.

VIEWMODEL: A ViewModel is a model for a view in the application or we can say an abstraction of the view. It makes data relevant to the view and behavior available to the views, usually with commands. The ViewModel is the link between the view and the outside world. The ViewModel is what the view is bound to. It provides a specialization of the model that the view can use for data binding.

The goal of MVVM is to separate the design of the application from the data and the functionality while supporting multiple development languages (e.g. C#/XAML) and taking full advantage of rich data binding.

Caliburn.Micro

In this small project we want to have Caliburn.Micro framework. Caliburn.Micro is a small but powerful framework designed for building applications on all XAML platforms. With strong support for MVVM and other proven UI patterns, Caliburn.Micro allows us to build our solution quickly without compromising code quality or testability. Caliburn.Micro uses a simple naming convention to locate Views for ViewModels. Essentially, it takes the FullName and removes "Model" from itTherefore, naming in MVVM is very important if you want to use Caliburn.Micro.

Logging

.NET supports a logging API that works with various built-in logging providers and third-party logging providers. Logging plays a crucial role when you cannot use interactive debugging. It is also used to monitor applications as an automatic tool that reports information about your application. This information includes application errors and user activity or requests. This is very different from logging, where your code actively writes messages and exceptions to the log. Logging can also be used to collect data and statistics about your users. This data can be used to research usage patterns, demographics, and behavior. There are 3Logging frameworks that pretty much dominate the .NET space. These are log4net, NLog and Serilog. Apache log4net is the oldest of the three frameworks. It was originally ported from the log4j project of Java. Serilog was the latest logging to join the party and provides a simple and easy to use solution for application logging to .NET application files. It is very easy to set up and has a clean API for logging. For these reasons, we decided to use Serilog.

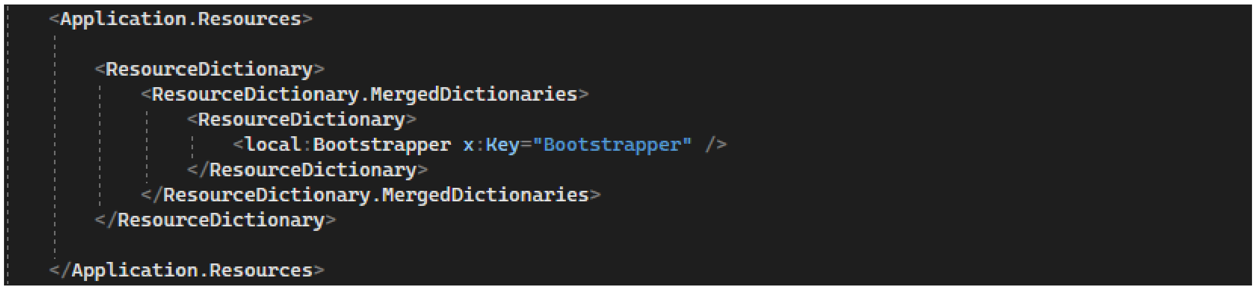
Dependency injection

Dependency injection is a form of the "Inversion of the Control" (IoC) programming principle. This means that classes do not create the objects they rely on. The main advantage of dependency injection is that the dependencies of all requested services are resolved before they are returned to the caller. DI frameworks have containers that are responsible for exposing and resolving dependencies. Dependency injection is not required for IOC containers to work, but it is a convenient way to decouple the consumer from the cached instances and from the cache itself.

IoC Simple Container: The term inversion of control means that the creation and keeping of the instance is no longer the responsibility of the consuming class, but is delegated to an external container. Caliburn.Micro comes preloaded with a dependency injection container called SimpleContainer. A dependency injection container is an object used to store dependency mappings for later use in an app via dependency injection. Dependency injection is actually a pattern that typically uses the container element instead of a manual service mapping.

Implementing Application

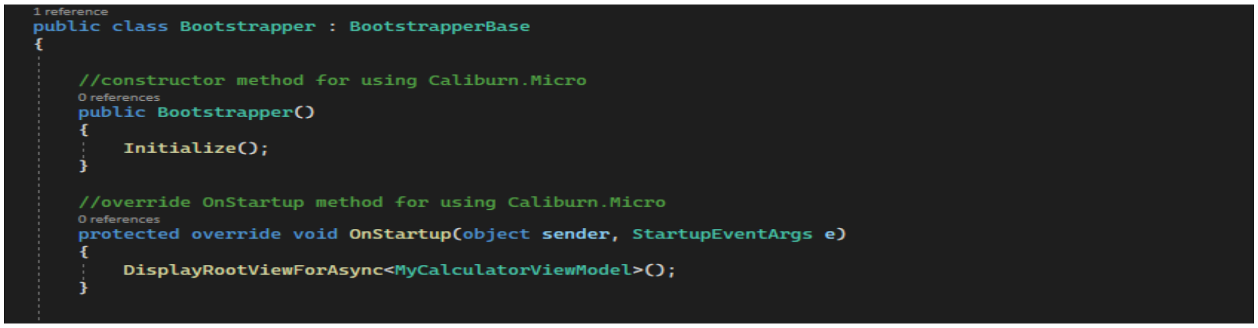
1. Create a .NET WPF application (.NET 6.0 LTS).
2. Delete MainWindow.xaml.
3. For the implementation of the WPF application with the **MVVM** pattern we should have 3 folders: View, ViewModel and Model. Thus, we created these three folders in WPF Calculator application.
4. In the **View** folder we have added a window called MyCalculatorView.xaml. This will be displayed to the user when running the application that contains the XAML codes to design the user interface.
5. In the **ViewModel** folder, we have a class called MyCalculatorViewModel.cs. This class is like a behind code in C# applications.
6. The **Model** folder, should contain all classes and interfaces. For example, in the calculator app there is a class called MyCalculatorModel.cs inherited from an interface named IMathCalculator.cs.
7. We have installed **Caliburn.Micro** via the Nuget package manager.
8. In App.xaml we deleted StartupUri="MainWindow.xaml" because we wanted Caliburn.Micro took care of the application startup.
9. The following tags have been added to App.xaml (Figure.1).



```
<Application.Resources>

    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary>
                <local:Bootstrapper x:Key="Bootstrapper" />
            </ResourceDictionary>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>

</Application.Resources>
```

Figure.1

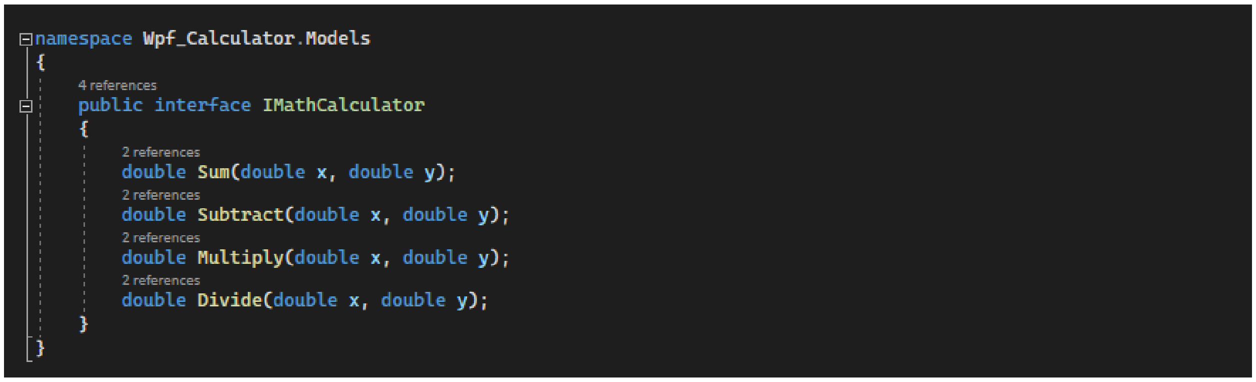
10. The Bootstrapper class was created, which inherited the BootstrapperBase class.
11. The following codes have been added to Bootstrapper class (Figure.2).



```
1 reference
public class Bootstrapper : BootstrapperBase
{

    //constructor method for using Caliburn.Micro
    0 references
    public Bootstrapper()
    {
        Initialize();
    }

    //override OnStartup method for using Caliburn.Micro
    0 references
    protected override void OnStartup(object sender, StartupEventArgs e)
    {
        DisplayRootViewForAsync<MyCalculatorViewModel>();
    }
```

Figure.2

12. This using Caliburn.Micro; should be added to Bootstrapper class.
13. In IMathCalculator interface, the following codes are added (Figure.3).

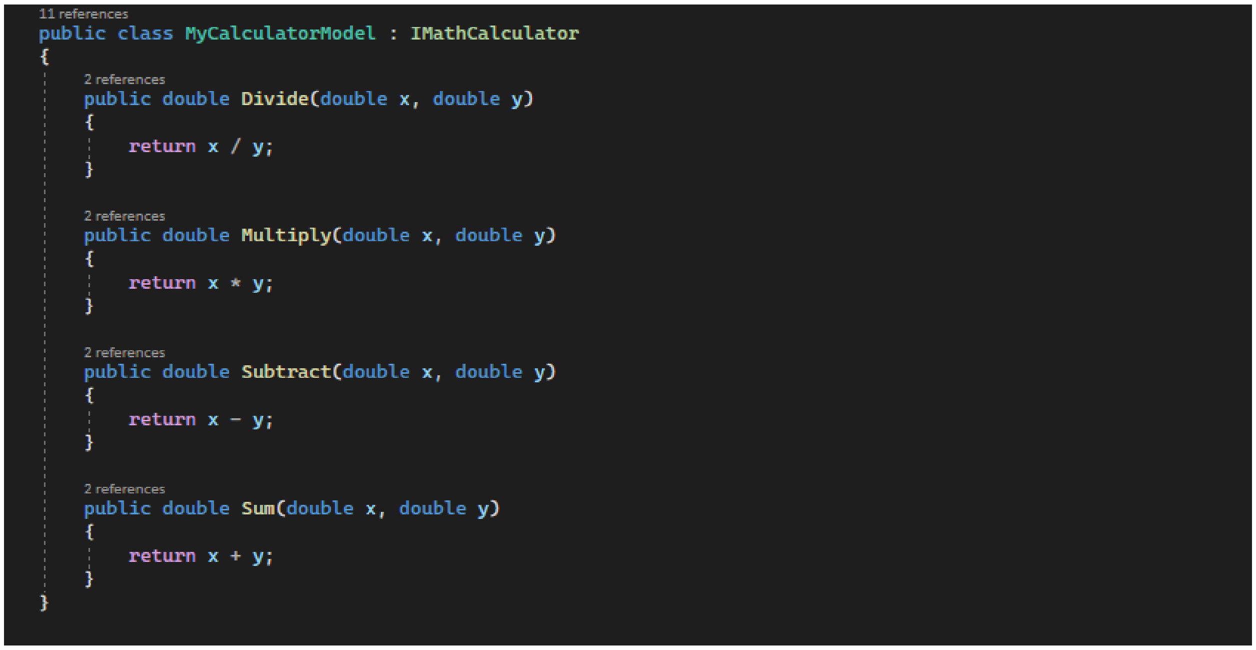


```
namespace Wpf_Calculator.Models
{
    4 references
    public interface IMathCalculator
    {
        2 references
        double Sum(double x, double y);
        2 references
        double Subtract(double x, double y);
        2 references
        double Multiply(double x, double y);
        2 references
        double Divide(double x, double y);
    }
}
```

Figure.3

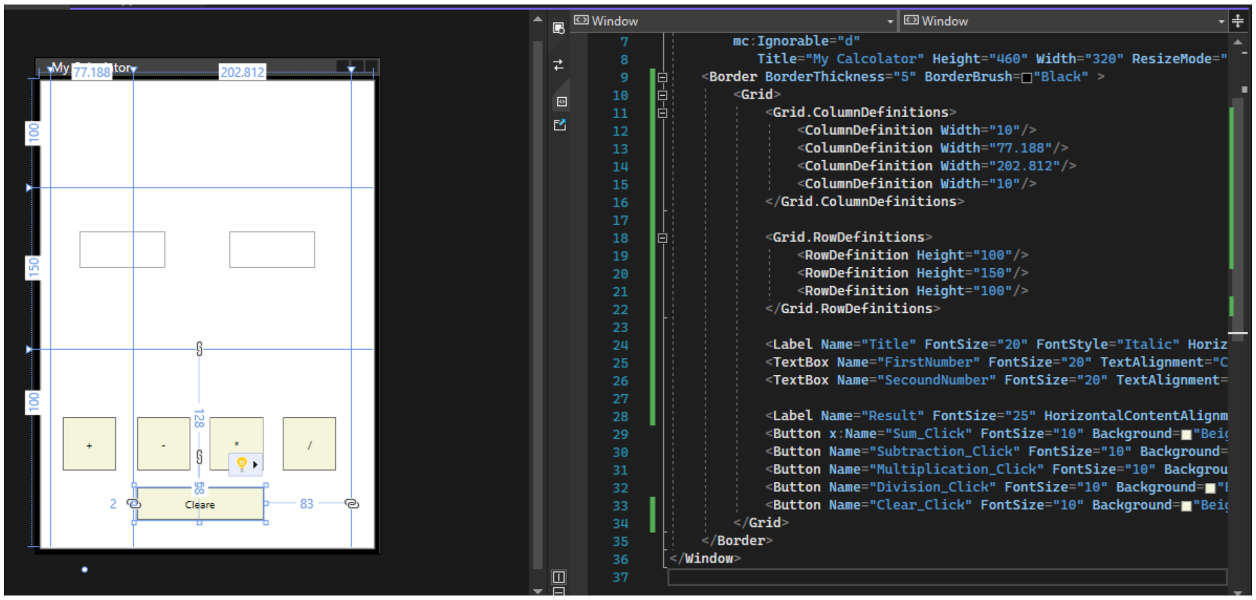
We used interfaces because using interface-based design concepts provides loose coupling, component-based programming, easier maintainability, makes our code base more scalable, and makes code reuse much more accessible because the implementation is separate from the interface.

14. When MyCalculatorModel class is inherited from IMathCalculator interface, all methods are completed by the following codes (Figure.4).
15. In MyCalculatorView.xaml we have added all UI controls like label, textbox, buttons with XAML language (Figure.5). Figure.6 shows the runtime view.



```
11 references
public class MyCalculatorModel : IMathCalculator
{
    2 references
    public double Divide(double x, double y)
    {
        return x / y;
    }

    2 references
    public double Multiply(double x, double y)
    {
        return x * y;
    }

    2 references
    public double Subtract(double x, double y)
    {
        return x - y;
    }

    2 references
    public double Sum(double x, double y)
    {
        return x + y;
    }
}
```

Figure.4



```
            mc:Ignorable="d"
                Title="My Calcolator" Height="460" Width="320" ResizeMode="
    <Border BorderThickness="5" BorderBrush=■"Black" >
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="10"/>
                <ColumnDefinition Width="77.188"/>
                <ColumnDefinition Width="202.812"/>
                <ColumnDefinition Width="10"/>
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="100"/>
                <RowDefinition Height="150"/>
                <RowDefinition Height="100"/>
            </Grid.RowDefinitions>

            <Label Name="Title" FontSize="20" FontStyle="Italic" Horiz
            <TextBox Name="FirstNumber" FontSize="20" TextAlignment="C
            <TextBox Name="SecoundNumber" FontSize="20" TextAlignment=

            <Label Name="Result" FontSize="25" HorizontalContentAlignm
            <Button x:Name="Sum_Click" FontSize="10" Background=□"Beig
            <Button Name="Subtraction_Click" FontSize="10" Background=
            <Button Name="Multiplication_Click" FontSize="10" Backgrou
            <Button Name="Division_Click" FontSize="10" Background=□"B
            <Button Name="Clear_Click" FontSize="10" Background=□"Beig
        </Grid>
    </Border>
</Window>
```

Figure.5

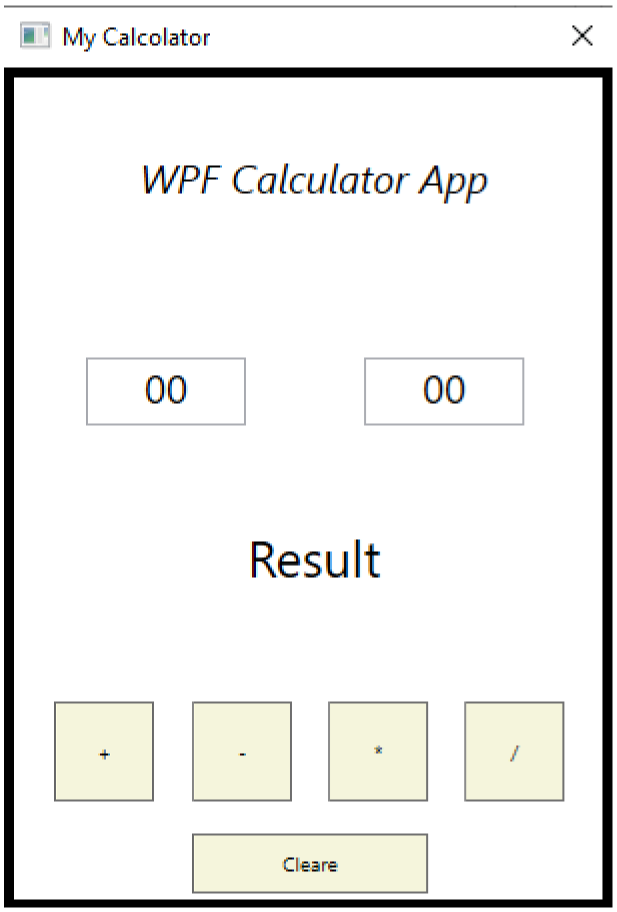
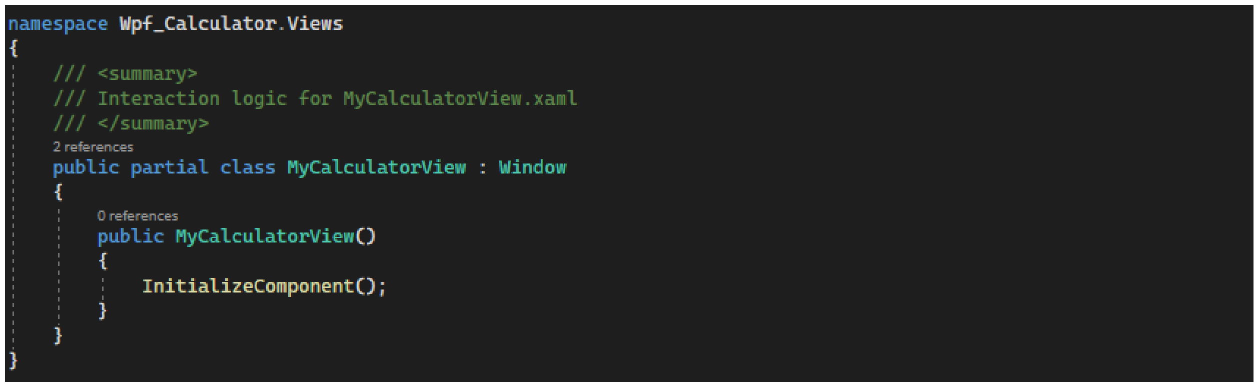


Figure.6

16. In MyCalculatorView.xaml.cs, the following codes are added (Figure.7):

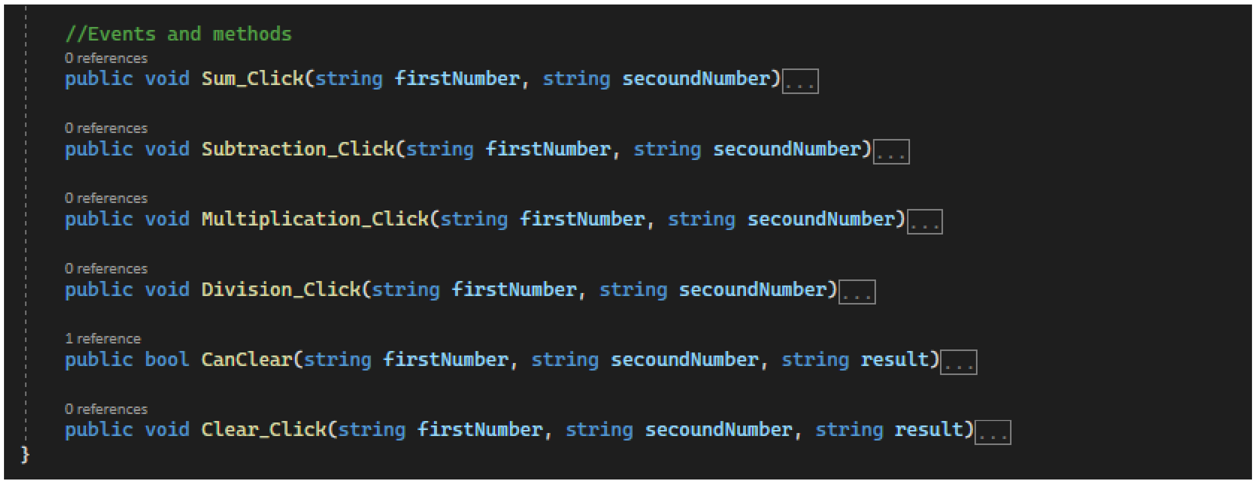


```
namespace Wpf_Calculator.Views
{
    /// <summary>
    /// Interaction logic for MyCalculatorView.xaml
    /// </summary>
    2 references
    public partial class MyCalculatorView : Window
    {
        0 references
        public MyCalculatorView()
        {
            InitializeComponent();
        }
    }
}
```

Figure.7

17. As we know, ViewModel has the objects that provide the data and functions for each of our views. In general, there is typically a one-to-one mapping between views and view model classes. The view model class makes the data available to the view and provides commands to handle user interaction. Unlike other design

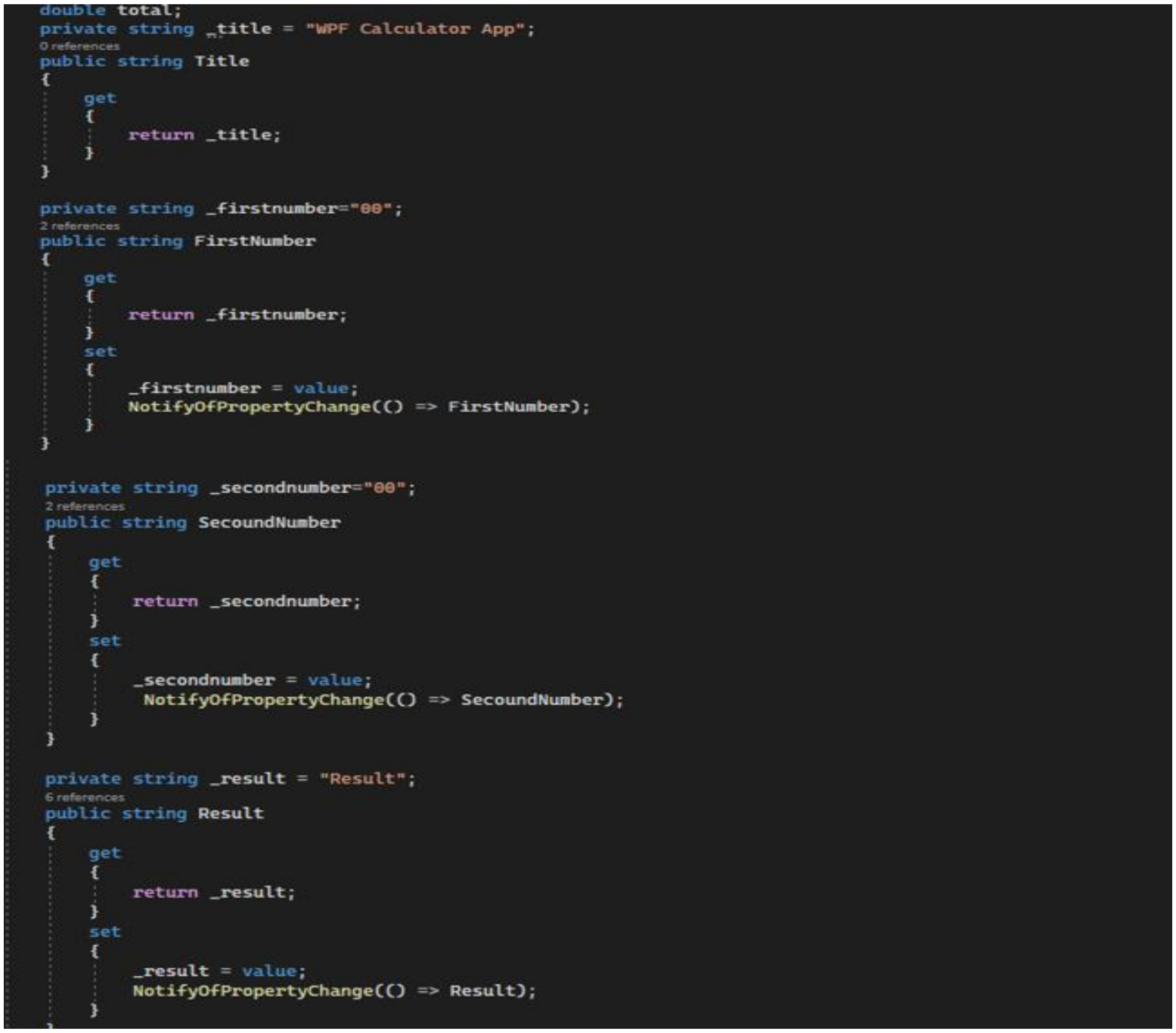
patterns, the view model should not know anything about its view. This separation of concerns is one of the key principles of MVVM. The view model is the link between the view and the model.
Figure.9 shows the codes are added to the MyCalculatorViewModel class. For all the controllers we have in View, we should have properties here, also all the events that happen when a button is clicked are added here (Figure.8).
Also, Caliburn.Micro contains several functions, and one of them is to simplify the implementation of the INotifyPropertyChanged interface. Notice that we added a NotifyOfPropertyChange call in the property's setter, which tells the view to display the new value whenever the property changes. This saves us from having to implement the INotifyPropertyChanged interface in all of our ViewModels (Figure.9).



```
    //Events and methods
    0 references
    public void Sum_Click(string firstNumber, string secoundNumber)...

    0 references
    public void Subtraction_Click(string firstNumber, string secoundNumber)...

    0 references
    public void Multiplication_Click(string firstNumber, string secoundNumber)...

    0 references
    public void Division_Click(string firstNumber, string secoundNumber)...

    1 reference
    public bool CanClear(string firstNumber, string secoundNumber, string result)...

    0 references
    public void Clear_Click(string firstNumber, string secoundNumber, string result)...
}
```

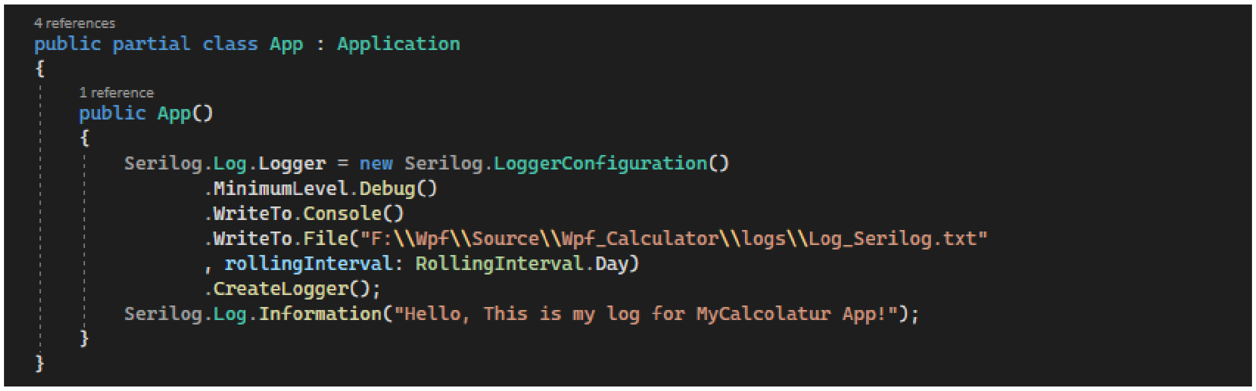
Figure.8

18. To implement **Serilog** the following three packages should be installed via NuGet Package Manager: 1. Serilog 2. Serilog.Sinks.Console 3. Serilog.Sinks.File. Then the following codes are added to App.xaml.cs (Figure.10).
19. Then we could use the Serilog wherever we needed it. Each day, a log file is placed in the previously defined location, with the current date added to the end of the log file. Figure.11 shows an example of how Serilog is used in this application. This sum_click event occurs when the user clicks on the Sum button. Figure.12 shows an example of Serilog written to the log file in our application.



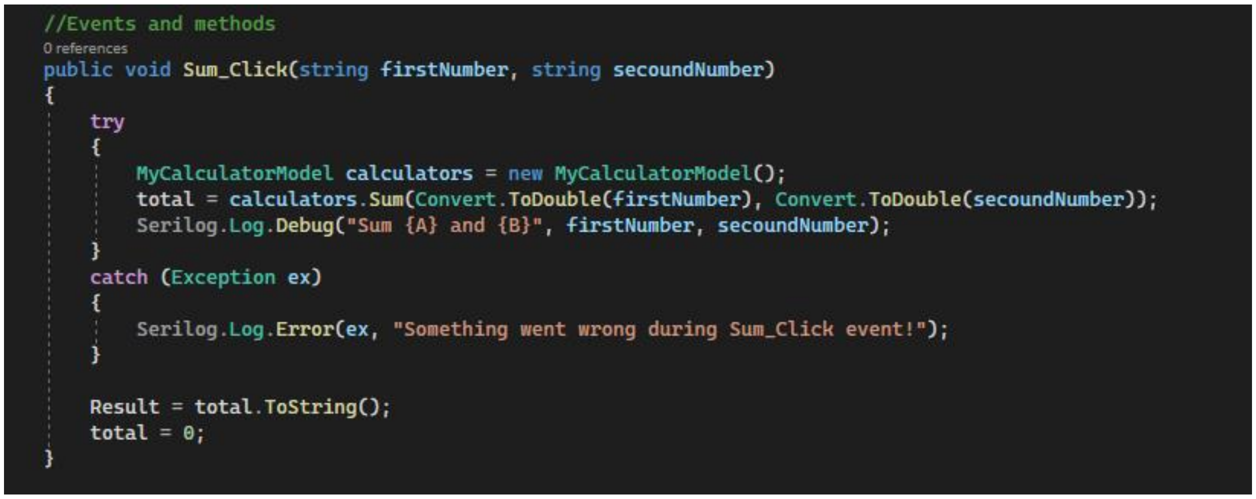
```
double total;
private string _title = "WPF Calculator App";
0 references
public string Title
{
    get
    {
        return _title;
    }
}

private string _firstnumber="00";
2 references
public string FirstNumber
{
    get
    {
        return _firstnumber;
    }
    set
    {
        _firstnumber = value;
        NotifyOfPropertyChange(() => FirstNumber);
    }
}

private string _secondnumber="00";
2 references
public string SecoundNumber
{
    get
    {
        return _secondnumber;
    }
    set
    {
        _secondnumber = value;
         NotifyOfPropertyChange(() => SecoundNumber);
    }
}

private string _result = "Result";
6 references
public string Result
{
    get
    {
        return _result;
    }
    set
    {
        _result = value;
        NotifyOfPropertyChange(() => Result);
    }
```

Figure.9



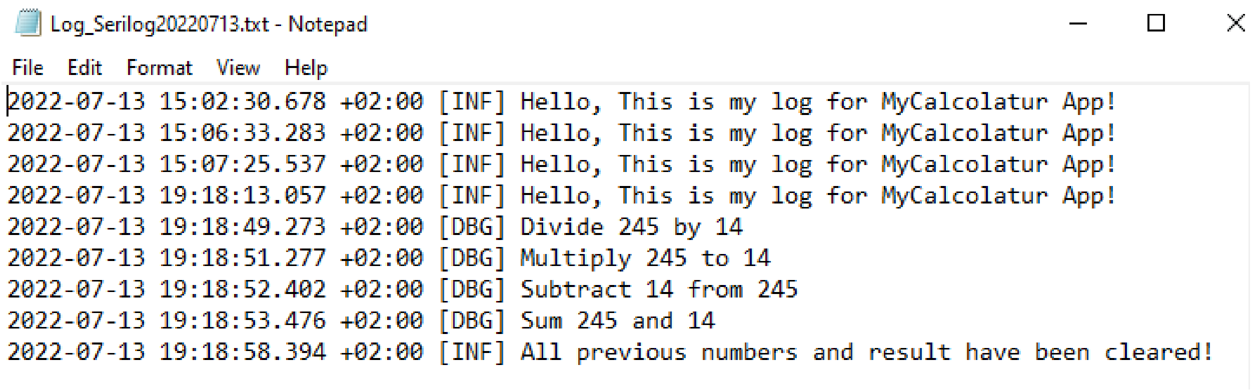
```
4 references
public partial class App : Application
{
    1 reference
    public App()
    {
        Serilog.Log.Logger = new Serilog.LoggerConfiguration()
                .MinimumLevel.Debug()
                .WriteTo.Console()
                .WriteTo.File("F:\\Wpf\\Source\\Wpf_Calculator\\logs\\Log_Serilog.txt"
                , rollingInterval: RollingInterval.Day)
                .CreateLogger();
        Serilog.Log.Information("Hello, This is my log for MyCalcolatur App!");
    }
}
```

Figure.10



```
//Events and methods
0 references
public void Sum_Click(string firstNumber, string secoundNumber)
{
    try
    {
        MyCalculatorModel calculators = new MyCalculatorModel();
        total = calculators.Sum(Convert.ToDouble(firstNumber), Convert.ToDouble(secoundNumber));
        Serilog.Log.Debug("Sum {A} and {B}", firstNumber, secoundNumber);
    }
    catch (Exception ex)
    {
        Serilog.Log.Error(ex, "Something went wrong during Sum_Click event!");
    }

    Result = total.ToString();
    total = 0;
}
```

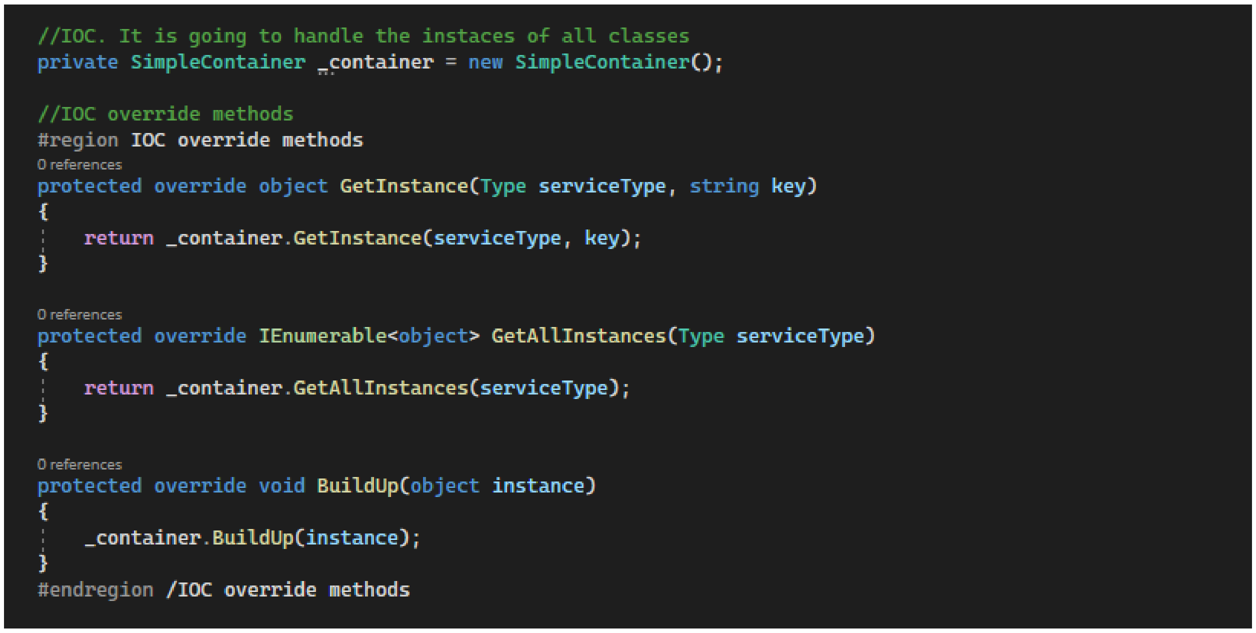
Figure.11



```
2022-07-13 15:02:30.678 +02:00 [INF] Hello, This is my log for MyCalcolatur App!
2022-07-13 15:06:33.283 +02:00 [INF] Hello, This is my log for MyCalcolatur App!
2022-07-13 15:07:25.537 +02:00 [INF] Hello, This is my log for MyCalcolatur App!
2022-07-13 19:18:13.057 +02:00 [INF] Hello, This is my log for MyCalcolatur App!
2022-07-13 19:18:49.273 +02:00 [DBG] Divide 245 by 14
2022-07-13 19:18:51.277 +02:00 [DBG] Multiply 245 to 14
2022-07-13 19:18:52.402 +02:00 [DBG] Subtract 14 from 245
2022-07-13 19:18:53.476 +02:00 [DBG] Sum 245 and 14
2022-07-13 19:18:58.394 +02:00 [INF] All previous numbers and result have been cleared!
```

Figure.12

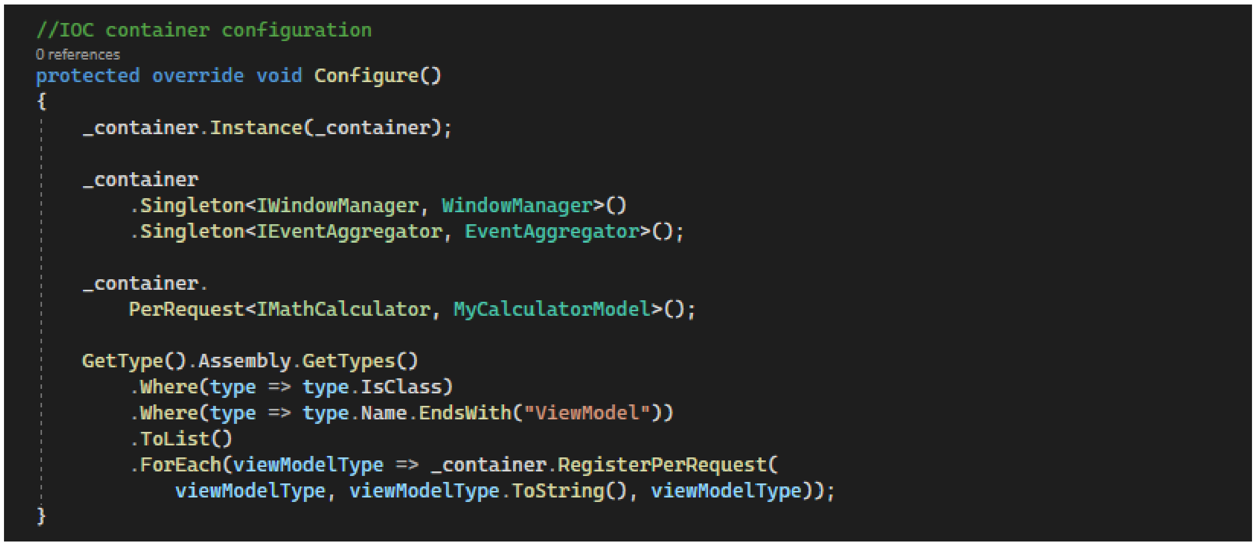
20. Before adding SimpleContainer service bindings, the container itself must be registered with Caliburn.Micro so that the framework can use the bindings. This process injects SimpleContainer into **IoC**, Caliburn.Micro's built-in service locator. The following code describes how to register SimpleContainer with Caliburn.Micro (Figure.12). We have added these codes to the bootstrapper class.



```
//IOC. It is going to handle the instaces of all classes
private SimpleContainer _container = new SimpleContainer();

//IOC override methods
#region IOC override methods
0 references
protected override object GetInstance(Type serviceType, string key)
{
    return _container.GetInstance(serviceType, key);
}

0 references
protected override IEnumerable<object> GetAllInstances(Type serviceType)
{
    return _container.GetAllInstances(serviceType);
}

0 references
protected override void BuildUp(object instance)
{
    _container.BuildUp(instance);
}
#endregion /IOC override methods
```

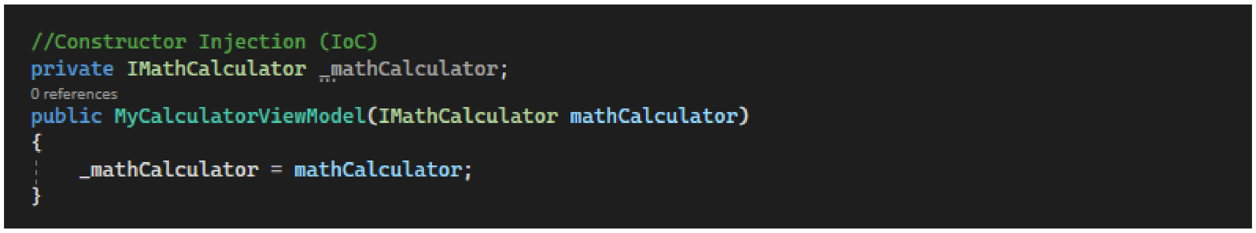
Figure.12

21. We override the Configure method for using IoC in our application. _container is an instance of the SimpleContainer class and is therefore used as an instance in the Configure method. The Singleton pattern is a convenient way to create an instance of a class and make it available. Instances of singleton registrations are created only when they are requested for the first time. Then we used it for our MyCalculatorModel, which inherited the IMathCalculator interface when we were asked to create a new instance of that class (Figure.13).



```
//IOC container configuration
0 references
protected override void Configure()
{
    _container.Instance(_container);

    _container
        .Singleton<IWindowManager, WindowManager>()
        .Singleton<IEventAggregator, EventAggregator>();

    _container.
        PerRequest<IMathCalculator, MyCalculatorModel>();

    GetType().Assembly.GetTypes()
        .Where(type => type.IsClass)
        .Where(type => type.Name.EndsWith("ViewModel"))
        .ToList()
        .ForEach(viewModelType => _container.RegisterPerRequest(
            viewModelType, viewModelType.ToString(), viewModelType));
}
```

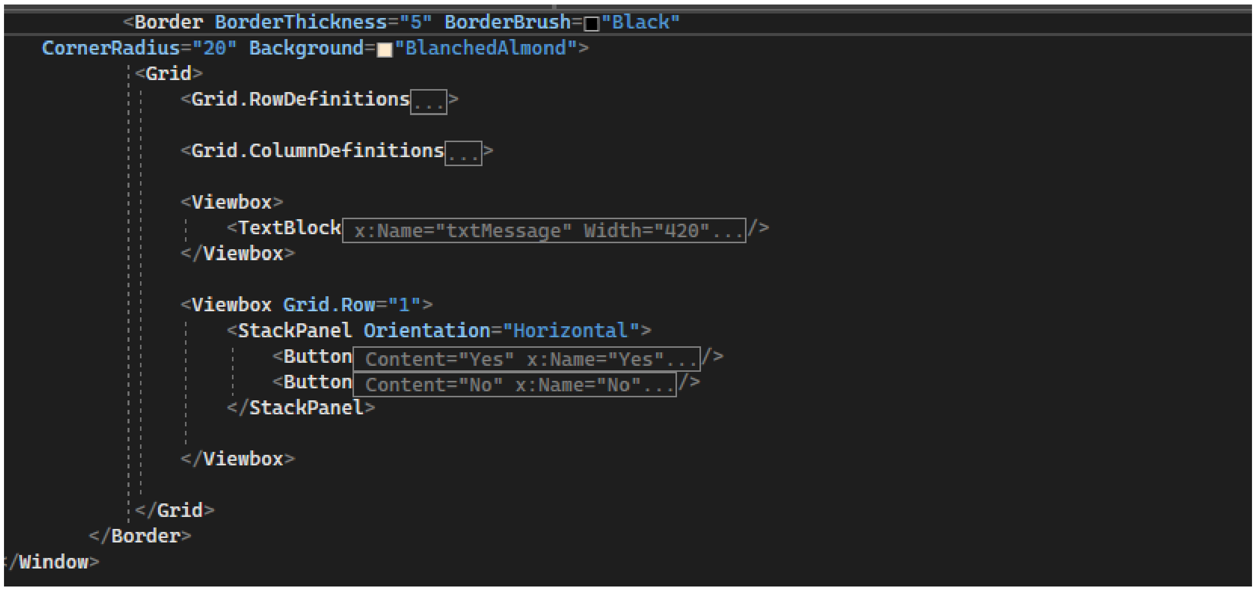
Figure.13



```
//Constructor Injection (IoC)
private IMathCalculator _mathCalculator;
0 references
public MyCalculatorViewModel(IMathCalculator mathCalculator)
{
    _mathCalculator = mathCalculator;
}
```

Figure.14

22. Constructor injection is the most common form of dependency injection and denotes a required dependency between services and the class into which they are inserted. The constructor injection should be used when you need the non-optional use of a particular service. (Figure.14).
23. We added a folder named Dialogs and 3 more folders Alert, Service and YesNo. Then a window named YesNoDialogBox.xaml was created in the YesNo folder. Figure.15 shows the codes that belong to YesNoDialogBox. Figure.16 shows the **dialog window** when the "Clear" button is clicked. Figure.17 shows YesNoDialogBox.xaml.cs code to implement constructor and methods. Figure.18 shows how we used this class in our application.



```
          <Border BorderThickness="5" BorderBrush="Black"
   CornerRadius="20" Background="BlanchedAlmond">
           <Grid>
               <Grid.RowDefinitions...>

               <Grid.ColumnDefinitions...>

               <Viewbox>
                   <TextBlock x:Name="txtMessage" Width="420"... />
               </Viewbox>

               <Viewbox Grid.Row="1">
                   <StackPanel Orientation="Horizontal">
                       <Button Content="Yes" x:Name="Yes"... />
                       <Button Content="No" x:Name="No"... />
                   </StackPanel>

               </Viewbox>

           </Grid>
       </Border>
</Window>
```

Figure.15

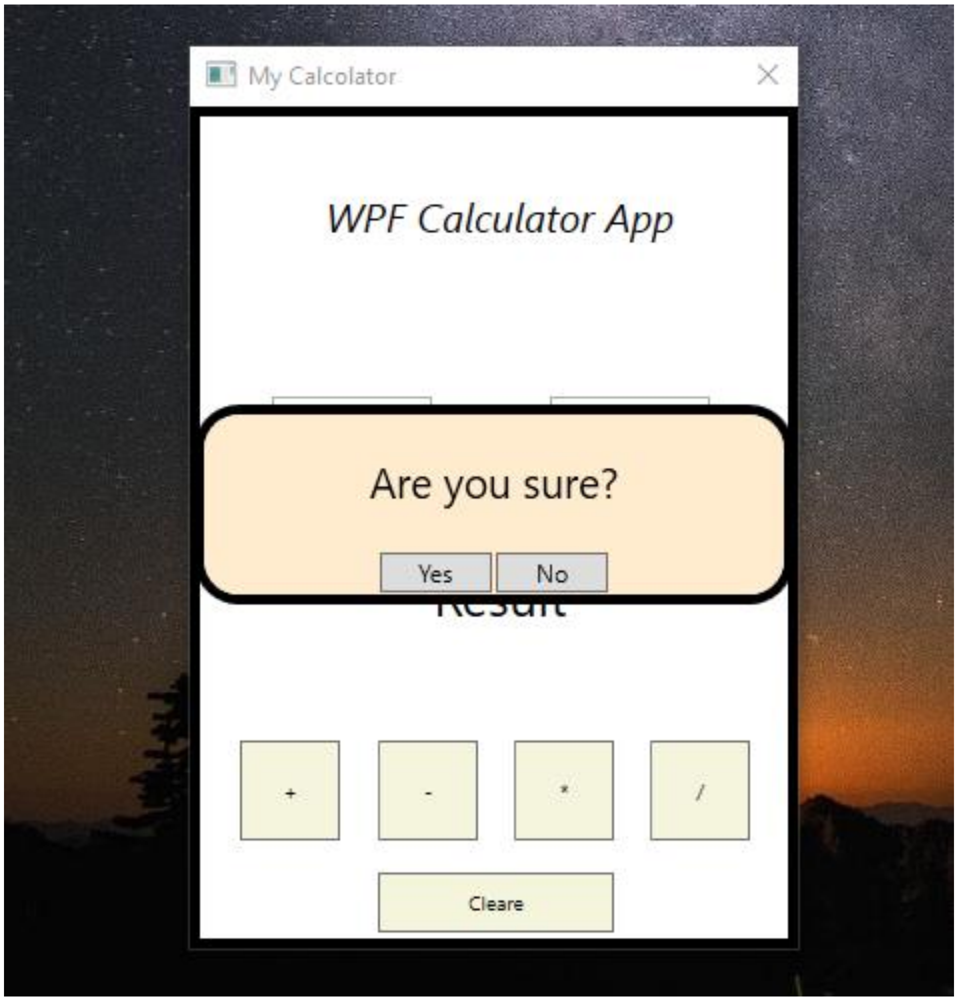
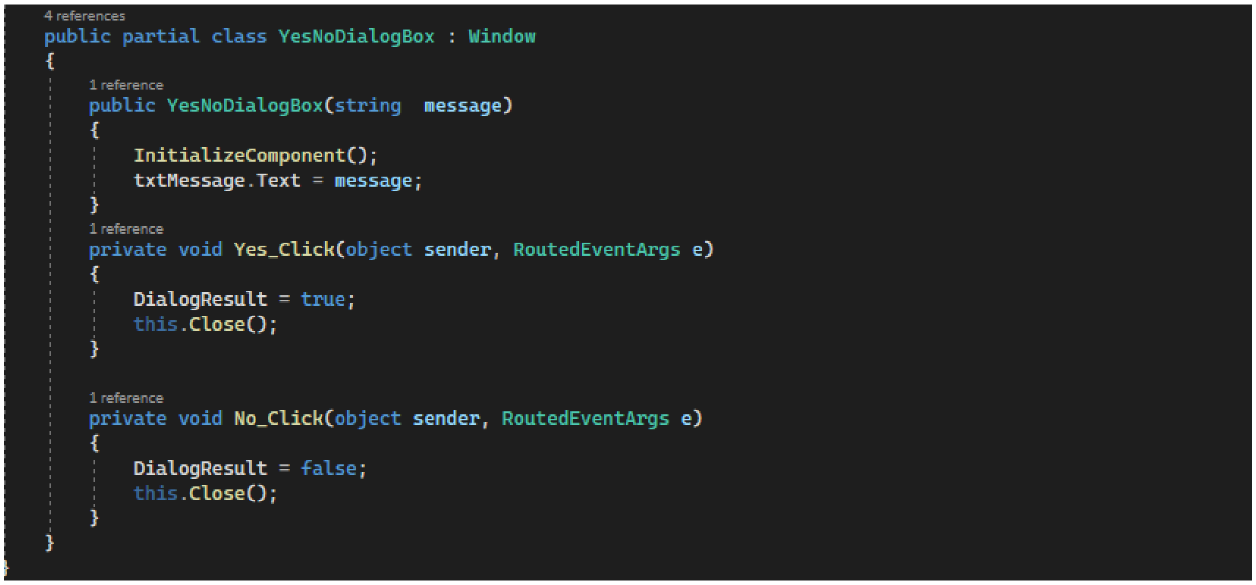
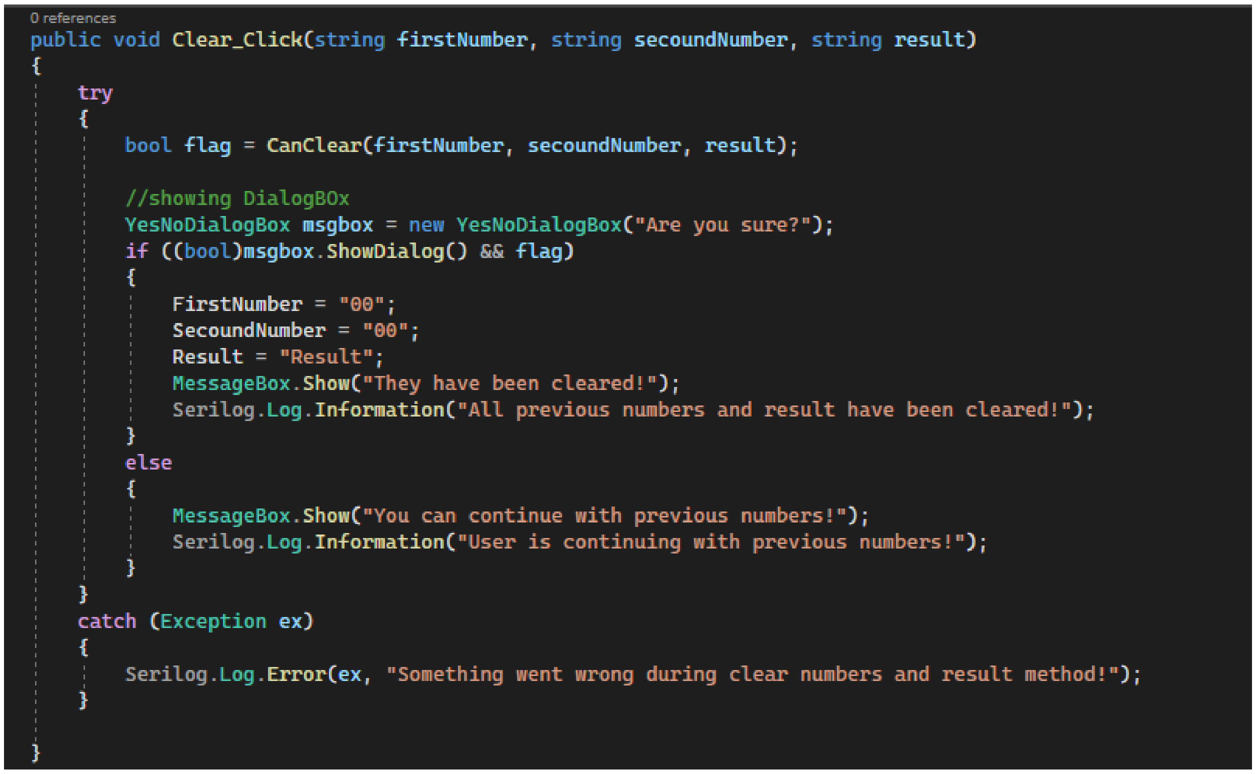


Figure.16



```
4 references
public partial class YesNoDialogBox : Window
{
    1 reference
    public YesNoDialogBox(string  message)
    {
        InitializeComponent();
        txtMessage.Text = message;
    }
    1 reference
    private void Yes_Click(object sender, RoutedEventArgs e)
    {
        DialogResult = true;
        this.Close();
    }

    1 reference
    private void No_Click(object sender, RoutedEventArgs e)
    {
        DialogResult = false;
        this.Close();
    }
}
```

Figure.17



```
0 references
public void Clear_Click(string firstNumber, string secoundNumber, string result)
{
    try
    {
        bool flag = CanClear(firstNumber, secoundNumber, result);

        //showing DialogBOx
        YesNoDialogBox msgbox = new YesNoDialogBox("Are you sure?");
        if ((bool)msgbox.ShowDialog() && flag)
        {
            FirstNumber = "00";
            SecoundNumber = "00";
            Result = "Result";
            MessageBox.Show("They have been cleared!");
            Serilog.Log.Information("All previous numbers and result have been cleared!");
        }
        else
        {
            MessageBox.Show("You can continue with previous numbers!");
            Serilog.Log.Information("User is continuing with previous numbers!");
        }
    }
    catch (Exception ex)
    {
        Serilog.Log.Error(ex, "Something went wrong during clear numbers and result method!");
    }

}
```

Figure.18

Two notes:

- 1. All classes should be public classes.
- 2. Before running, the app should be built and executed first to avoid the occurrence of errors related to the bootstrapper class.