MVVM Demonstration Using C# WPF

Sudip Chakraborty [1] & P. S. Aithal [2]

[1] D.Sc. Researcher, Institute of Computer Science and Information sciences, Srinivas University, Mangalore-575 001, India,
OrcidID: 0000-0002-1088-663X; E-mail: sudip.pdf@srinivasuniversity.edu.in

[2] Vice Chancellor, Srinivas University, Mangalore, India,
OrcidID: 0000-0002-4691-8736; E-Mail: psaithal@gmail.com

Subject Area: Computer Science.
Type of the Paper: Experimental Research.
Type of Review: Peer Reviewed as per |C|O|P|E| guidance.
Indexed In: OpenAIRE.
DOI: https://doi.org/10.5281/zenodo.7538711
Google Scholar Citation: IJAEML

How to Cite this Paper:
Chakraborty, S., & Aithal, P. S., (2023). MVVM Demonstration Using C# WPF. *International Journal of Applied Engineering and Management Letters (IJAEML)*, *7*(1), 1-14. DOI: https://doi.org/10.5281/zenodo.7538711

International Journal of Applied Engineering and Management Letters (IJAEML)
A Refereed International Journal of Srinivas University, India.

Crossref DOI: https://doi.org/10.47992/IJAEML.2581.7000.0163

Received on: 12/12/2022
Published on: 14/01/2023

MVVM Demonstration Using C# WPF

Sudip Chakraborty [1] & P. S. Aithal [2]

[1] D.Sc. Researcher, Institute of Computer Science and Information sciences, Srinivas University, Mangalore-575 001, India,

OrcidID: 0000-0002-1088-663X; E-mail: sudip.pdf@srinivasuniversity.edu.in

[2] Vice Chancellor, Srinivas University, Mangalore, India,

OrcidID: 0000-0002-4691-8736; E-Mail: psaithal@gmail.com

ABSTRACT

Purpose: Nowadays, MVVM (Model-View View-Model) is the proven architecture for distributed software development. It encourages the development of the software components by the different independent teams and easy integration at the final stage. The individual researcher prefers direct coding or tightly coupled software modules. The Model is rapid and efficient but might create maintainability issues later. So from day one, we should introduce the best design model. In this scenario, the MVVM model is the major player. Here, we present how to implement MVVM into our project through a simple task. We will design the user interface, which is part of the UX design team, and then add functionality, which is the programming part. Finally, we will do integration and execution. The code used in this project is available to download from GitHub.

Design/Methodology/Approach: We are creating a C# WPF project inside the visual studio community edition. Then we segregate our activity into two parts. In the first part, we make a Model for our project. After that, we design the user interface. The user interface interacts with the user to display the data and receive inputs from the user. It is the presentation or view layer. After completing it, we add the required view model interaction logic. Finally, we integrate all components and run the project.

Findings/Result: Through the research, we realize the importance of the MVVM concept. It is a good software architecture; the researcher who has gone through the documents will find out how to implement the MVVM into their project. Some essential procedures are presented concisely so they can be adopted quickly. This architecture is independent of any language. So once we grab it, it can be implemented in our project, whatever the language we use.

Originality/Value: Several documents on MVVM design using WPF are available worldwide. Most of the documents are elaborative and descriptive. It is tricky to extract the required information from long-duration content as fast-track space. Here we demonstrate practically. Using the documents as a reference, the researcher can easily integrate the MVVM concept into their project.

Paper Type: Experimental-based Research.

Keywords: MVVM concept, MVVM design, Model View Model Design, MVVM in C# WPF.

1. INTRODUCTION :

Day by day, we are surrounded by more and more electronic gadgets. The gadget's behaviour is also becoming more complex. A few years ago, we developed the software, and it would run year after year without an issue, and no change requirement was there. Now the scenario has changed. That robust software we built just a few months back is outdated for various reasons.

(1) We are now operating the application from various devices like desktops, laptops, tablets, Mobile phones, etc. They are all different screen resolutions. The application's target screen resolution is dynamic. So we need to develop our apps with auto-adjustable resolution features.

(2) The market is more competitive, and the cost of production is generally so high, so we need cheap and the best solution in every aspect.

(3) We increasingly understand the user and user usage patterns and update our apps, and we want to stay away from the competition.

(4) The development time of the projects is too short now. Sometimes we must launch a new software product in a couple of weeks.
(5) The marketing philosophy has drastically changed. Instead of a static image, one short-duration video or animated movie is preferable for the user to understand the content. The device computation power is also increasing, so we must deliver dynamic content instead of static content.
(6) The simple windows form with a 3D effect border or background is more resource consumable.

Instead, flat architecture with high responsiveness is effective and widely acceptable. So, software technologies have changed their strategy to the need. Now MVVM technology has come. MVVM stands for model view model architecture. It provides loose binding among various software components. So that one wrong implementation does not adversely affect the entire software stack. That is why this architecture is so popular. The Model means the Model of the software product, and the view is the display item of the content to the user. We distribute our assigned projects to specialized teams and build high-quality software with a competitive advantage.
So the advantage of the MVVM architecture are:-

- It is a rapid prototyping model
- Distributed development architecture
- Loosely coupled among modules
- Better maintainability.
- Easy to debug

2. RELATED WORKS :

Pan, H. H et al. design their GUI interface architecture using MVVM in WPF. Their GUIAC has a loosely coupled structure with three layers, so implementation works parallel to improve development and maintenance efficiency [1]. Sorensen E et al. worked on education systems that are beginning to adapt and transform the electronic versions, marking the transition from the traditional to the digital Age [2]. Gross, I. V. et al. developed their application using .Net WPF user control and 3D picture box control for facilitating 3D image reconstruction [3]. Sulistyarini, D et al. design them using WPF for flexibility of different screen resolutions. They use SQL servers for data storage, and the result shows inside the WPF control [4]. Troelsen, A. et al. in their book dedicated a particular chapter to the investigation of the WPF programming model by covering the capabilities that support the Model-View-ViewModel (MVVM) pattern, which improves the user experience significantly and reduces the manual coding required in older technologies [5].

3. OBJECTIVES :

MVVM is an excellent architectural concept in the software design principle. It makes software projects easy to build and maintain. The primary objective is to introduce our researchers doing software projects using C# WPF. And also, the aim is to provide some good references or handy documents for the researcher to integrate MVVM into their project.

Figure 1 depicts the architecture of our project. When we developed our project using MVVM architecture, we divided our task into several teams. We can determine how to build our project using two teams' work. More division may be possible when a specialized workforce is available. The core function is two. The UX interface and programming task. Here we are describing how to execute our job step by step:
(1) We create a C# WPF Application in Microsoft visual studio community edition 2022. If we build and run the apps, blank windows will appear.
(2) Model: Create a folder Model and add the class *Employee.cs* and *EmployeeService.cs*. Inside the *Employee.cs* create variables with properties get/set. This is the data storage place. We save and retrieve the data from here. Inside the *EmployeeService.cs*, we add functionality to the operation on model data. It is the collection of methods to work on model data—task number 1 and 2 on the right side of figure 1.
(3) View: The project requirement generated by the client or from the administration. The technical writer creates software requirement specifications (SRS). The UX team creates

layouts using tools like Figma apps. In figure 4.1, the left side 1 and 2 task indicates the scenario. Then GUI team develops the front end. Here we Create a folder called "**View**" and add a user control. Name "*EmployeeView.XAML*". According to our requirement, we add several controls like text boxes, buttons, sliders, combo boxes, etc. After the GUI design, we bind the UI element with methods and variables. It bridges two sides of the project design and programming. The methods we write inside the *EmployeeViewModel.cs* class depicts task 4 of the figure.

4. APPROACH AND METHODOLOGY :

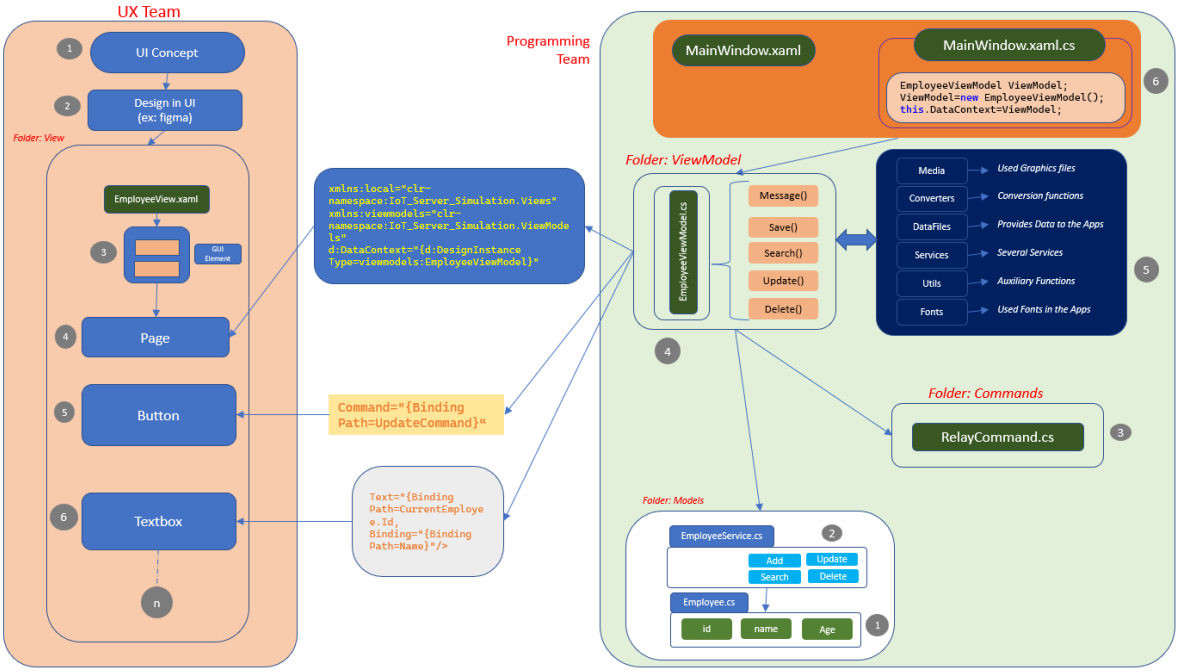

Fig. 1: Project Block Diagram

(4) Commands: This is used to relay the commands to the specific methods from the user interface events. It also helps to eliminate hard binding between UI and associated procedures. create a folder called "Command." Add a class *RelayCommand.cs*. Add methods to relay. We can see task 3 In figure 1.

(5) ViewModel: This part is the central part of the complete architecture. It coordinates among all components of the application. Here we Create a folder named "*ViewModel*," and inside it, we create a class "*EmployeeViewModel.cs*". When this class instantiates, it establishes several objects. One temporary object class that stores currently edited data are responsible for sending as a method parameter to methods like create, delete, etc. Next, develop functions that will be relayed to the specific techniques. And with this class, we establish several modules to support itself, like communication, database, etc., to provide the error-free content flow among modules which depicts task 5 of figure 1.

(6) App Entry Point: After all module development, one little bit of pending task is still to execute the application. We need to connect the Main windows with the coordinate class. In figure 1, task 6 needs to do. We initiate the class here so that it can navigate the entire application.

5. EXPERIMENT :

Now we will experiment with MVVM architecture, following the below steps. Here, we can type or copy the entire class from the downloaded project folder from GitHub to create any class.

Project creation: Create a repository in GitHub and Clone it. Open visual studio. Create a new C# WPF project and name it "**MVVM_Demo**." Build the project and run. It will show blank windows with white background.

Model Creation: Inside the project, create a folder named "Model." Inside the model folder, create two classes, "***Employee. cs***" and "***EmployeeService.cs***" write the code as depicted in figure 2 or copy-paste. Build the project. It should be a success.

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

namespace MVVM_Demo.Model
                                                    {
   public class Employee : INotifyPropertyChanged
   {
      public event PropertyChangedEventHandler PropertyChanged;

      private void OnPropertyChanged(string propertyName)
      {
         if (PropertyChanged !=null)
         {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
         }
      }
      private int id;
      public int Id
      {
         get {return id;}
         set{id=value;OnPropertyChanged("Id");}
      }
      private String name;
      public string Name
      {
         get { return name;}
         set { name=value; OnPropertyChanged("Name");}
      }
      private int age;
      public int Age
      {
         get{return age;}
         set{age=value;OnPropertyChanged("Age");}
      }

   }// class
}
```

Fig. 2: Example code for Employee.cs

Description: This is the primary or base class. The variable we work on within a project will be assigned or created here. Every variable is accessed through its property. When we create a variable, we also make its property so that we can save or retrieve it to and from the variable. Two things are essential here. One is a variable assignment, and another is an *INotifyPropertyChanged* assignment. This is one of the essential things for MVVM architecture. When the user changes the textbox content, the function will be called automatically and set property, save the value to the associated variable. If the control is two-way binding, it is held and loaded from this variable. We need to include "using System.ComponentModel" for it. We declare one event handler *PropertyChangedEventHandler*. When any one of the variable properties changes, it will interrupt, and the OnPropertyChanged method will be called. It is the standard function for all variables. When it is called, it will take the selected property and set or get the value from the variable.

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

namespace MVVM_Demo.Model
{
   public class EmployeeService
   {
      private static List<Employee> objEmployeesList;

      public EmployeeService()
      {
         objEmployeesList=new List<Employee>();
      }
      //_____________________________________________________________________
      public List<Employee> GetAll()
      {
         return objEmployeesList;
      }
      //_____________________________________________________________________
      public bool Add(Employee objNewEmployee)
      {
         // Age must between 21 and 58
         if (objNewEmployee.Age<21 || objNewEmployee.Age>58)
            throw new ArgumentException("Invalid age limit for employee");
                objEmployeesList.Add(objNewEmployee);
         return true;
      }
      //_____________________________________________________________________
      public bool Update(Employee objEmployeeToUpdate)
      {
         bool IsUpdate = false;
         for (int index = 0; index<objEmployeesList.Count; index++)
         {
            if (objEmployeesList[index].Id==objEmployeeToUpdate.Id)
            {
               objEmployeesList[index].Name=objEmployeeToUpdate.Name;
               objEmployeesList[index].Age=objEmployeeToUpdate.Age;
               IsUpdate=true;
               break;
            }
         }
         return IsUpdate;
      }
      //_____________________________________________________________________
      public bool Delete(int id)
      {
         bool IsDeleted = false;
         for (int index = 0; index<objEmployeesList.Count; index++)
         {
            if (objEmployeesList[index].Id==id)
            {
               try
               {
                  objEmployeesList.RemoveAt(index);
                  IsDeleted = true;
               }
               catch (Exception)
               {

               } } }
                  return IsDeleted;
      }
      //_____________________________________________________________
      public Employee Search(int id)
      {
         return objEmployeesList.FirstOrDefault(e => e.Id==id);
      }
      //_____________________________________________________________
   }
  }
```

Fig. 3: Example code for *EmployeeService.cs*

Here we created three variables, *id*, *name*, and *Age*, to store the data. Three text boxes are available in the view module, which will be displayed to the user. These three textboxes are bonded with these variables. When some change occurs, the event is fired, and the value is set or gets to and from the variable.

Now we will write the "*EmployeeService.cs*" class. We can write the code depicted in figure 5.2 or copy from the figure and paste the code inside the class.

Description: first, we create a list object of our base class, the *Employee* class. The *GetAll* function get all Employee as a list to the caller. The add function adds the new Employee. Before adding the Employee, we can check various criteria like age validation. If all requirements are valid, one new Employee will be added to the employee list object. The next one is Update methods. This method is

used to update or modify any information. We are passing the modified employee details. If the ID is matched, the Employee is updated. The Delete function is used to delete any records from our collection. Before deleting any form, we should check the ID of the Employee. And the last one is search methods. This method searches for Employee from the object collection. In a real-life project, we add more variables here.

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

namespace MVVM_Demo.Command
{
   public class RelayCommand:ICommand
   {
      public event EventHandler CanExecuteChanged;
      public Action DoWork;

      public RelayCommand(Action work)
      {
         DoWork= work;
      }

      public bool CanExecute(object parameter)
      {
         return true;
      }

      public void Execute(object parameter)
      {
         DoWork();
      }
   }
}
```

Fig. 4: Example code for *RelayCommand.cs*

Relay Command: Inside the project folder, create a folder called "***Command***." create a class called "***RelayCommand.cs***". Add some methods as depicted in figure 4. We can type or copy/paste using text selection. This command relay isolates the user interface among different modules.

Description: This class is implemented targeting isolation. When users press some control like a button or any other command input, the program pointer reaches here and then relays to the particular command. It acts as the bridge between the module. The modules are developed individually and connected virtually. It helps us to bind the modules loosely. If we write a direct function, that also works, but if some changes occur, both modules are affected. Here the *ICommand* is a runtime binding element, and the direct coding compiles time binding.

ViewModel Design: we create a folder called "***ViewModel***." Inside the folder, create a class "***EmployeeViewModel.cs***". We can copy the code or direct copy and paste it into the class.

```
using MVVM_Demo.Command;
using MVVM_Demo.Model;

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

namespace MVVM_Demo.ViewModel
{
   public class EmployeeViewModel : INotifyPropertyChanged
   {
      #region Inotify change
      public event PropertyChangedEventHandler PropertyChanged;
      private void OnPropertyChanged(string propertyName)
      {
         if (PropertyChanged !=null)
         {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
         }
      }
      #endregion
      EmployeeService objEmployeeService;
      public EmployeeViewModel()
      {
         objEmployeeService= new EmployeeService();
         LoadData();
         CurrentEmployee     =new Employee();
         SaveCommand=new RelayCommand(Save);
         searchCommand=new RelayCommand(Search);
         updateCommand=new RelayCommand(Update);
         deleteCommand=new RelayCommand(Delete);
      }
      #region Display Operation
      private ObservableCollection<Employee> employeesList;
      public ObservableCollection<Employee> EmployeesList
      {
         get {return employeesList;}
         set{employeesList = value; OnPropertyChanged("EmployeesList");}
      }
      private void LoadData()
      {
         EmployeesList=new ObservableCollection<Employee>(objEmployeeService.GetAll());
      }
      #endregion
      private Employee currentEmployee;
      public Employee CurrentEmployee
      {
         get {return currentEmployee;}
         set{ currentEmployee=value;OnPropertyChanged("CurrentEmployee");}
      }
      private string message;
      public string Message
      {
         get{return message;}
         set{message=value; OnPropertyChanged("Message");}
      }
      #region SaveOperation
      private RelayCommand saveCommand;

      public RelayCommand SaveCommand
      {
         get{ return saveCommand;}
         set{saveCommand = value;}
      }
      public void Save()
      {
         try
         {
            Employee emp = new Employee();
            emp.Id=CurrentEmployee.Id;
            emp.Name=CurrentEmployee.Name;
            emp.Age=CurrentEmployee.Age;

            var IsSaved = objEmployeeService.Add(emp);
            LoadData();
            if (IsSaved) Message="Employee Saved";
            else
               Message="Save Operation Failed";
         }
         catch (Exception ex)
         {
```

Fig. 5 (a): Example code for *EmployeeViewModel.cs*

```
            Message=ex.Message;
        }
    }

    #endregion

    #region SearchOperation
    private RelayCommand searchCommand;

    public RelayCommand SearchCommand
    {
        get
        {
            return searchCommand;
        }
        set
        {
            searchCommand = value;
        }
    }

    public void Search()
    {
        try
        {
            var ObjEmployee = objEmployeeService.Search(CurrentEmployee.Id);
            if (ObjEmployee!= null)
            {
                CurrentEmployee.Name= ObjEmployee.Name;
                CurrentEmployee.Age= ObjEmployee.Age;
            }
            else
            {
                Message="Employee Not found";
            }
        }
        catch (Exception ex)
        {

            throw;
        }
    }
    #endregion

    #region UpdateOperation
    private RelayCommand updateCommand;

    public RelayCommand UpdateCommand
    {
        get
        {
            return updateCommand;
        }
    }

    public void Update()
    {
        try
        {
            var IsUpdated = objEmployeeService.Update(CurrentEmployee);
            if (IsUpdated)
            {
                Message="Employee Updated";
                LoadData();
            }
            else
            {
                Message="Update Operation Failed";
            }
        }
        catch (Exception ex)
        {

            Message=ex.Message;
        }
    }
    #endregion


    #region DeleteOperation

    private RelayCommand deleteCommand;
```

Fig. 5(b): Example code for *EmployeeViewModel.cs (continue..)*

```
public RelayCommand DeleteCommand
{
   get
   {
      return deleteCommand;
   }
}

public void Delete()
```

Fig. 5(c): Example code for *EmployeeViewModel.cs (continue..)*

```
<UserControl x:Class="MVVM_Demo.View.EmployeeView"
        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/expression/blend/2008"
        xmlns:local="clr-namespace:MVVM_Demo.View"
         xmlns:viewmodels="WpfApp1.ViewModel"
        xmlns:viewmodel="clr-namespace:MVVM_Demo.ViewModel"
        d:DataContext="{d:DesignInstance Type=viewmodel:EmployeeViewModel}"


        mc:Ignorable="d"
        d:DesignHeight="600" d:DesignWidth="800">
  <Grid Margin="15" >
    <Grid.ColumnDefinitions >
      <ColumnDefinition Width="23.582" />
      <ColumnDefinition Width="38.778"/>
      <ColumnDefinition Width="26*" />
      <ColumnDefinition Width="323*"/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
      <RowDefinition Height="96*" />
      <RowDefinition Height="96*" />
      <RowDefinition Height="96*" />
      <RowDefinition Height="96*" />
      <RowDefinition Height="96*" />
      <RowDefinition Height="90*" />
      <RowDefinition Height="101*" />
    </Grid.RowDefinitions>

    <TextBlock Text="Employee Management"
           Grid.Row="0"
           Grid.Column="3"
           FontSize="20"
           FontWeight="Bold"
           HorizontalAlignment="Left" Margin="155,0,0,53" Grid.RowSpan="2" Width="222"
     />

    <TextBlock Text="Enter Id"
           Grid.Column="0" Grid.ColumnSpan="2" Margin="0,68,0,96" Grid.RowSpan="2" />
    <TextBlock Text="Enter Name"
           Grid.Row="1"
           Grid.Column="0" Grid.ColumnSpan="2" Margin="0,15,0,53" />
    <TextBlock Text="Enter Age"
           Grid.Row="1"
           Grid.Column="0" Grid.ColumnSpan="2" Margin="0,63,0,96" Grid.RowSpan="2" />

    <TextBox Name="txtId"
           Grid.Column="2" Grid.ColumnSpan="2" Margin="10,68,438,96"
          Text="{Binding Path=CurrentEmployee.Id,Mode=TwoWay}" Grid.RowSpan="2"

         />
    <TextBox Name="txtName"
           Grid.Row="1"
           Grid.Column="2" Grid.ColumnSpan="2" Margin="11,10,437,53"
         Text="{Binding Path=CurrentEmployee.Name,Mode=TwoWay}"
         />

    <TextBox Name="txtAge"
           Grid.Row="1"
           Grid.Column="2" Grid.ColumnSpan="2" Margin="11,56,438,96"
          Text="{Binding Path=CurrentEmployee.Age,Mode=TwoWay}" Grid.RowSpan="2"
         />


    <StackPanel Orientation="Horizontal"
```

Fig. 6: Example code for *EmployeeView. XAML*

```
            Grid.Row="2"
            Grid.Column="2" Grid.ColumnSpan="2" Margin="5,9,-6,19">
        <Button Name="btnAdd"
                Content="ADD"
              Margin="5,8" Padding="3"
              Width="113" Height="50"
                 Command="{Binding Path=SaveCommand}" RenderTransformOrigin="0.428,-0.45"
              />

        <Button Name="btnSearch"
                 Content="SEARCH"
                Margin="5,8" Padding="3"
               Width="97" Height="50"
                 Command="{Binding Path=SearchCommand}"
              />
        <Button Name="btnUpdate"
                 Content="UPDATE"
                Margin="5,8" Padding="3"
               Width="108" Height="50"
                 Command="{Binding Path=UpdateCommand}"
              />
        <Button x:Name="btnDelete"
                 Content="DELETE"
               Padding="3"
                Width="80" Height="50"
                Command="{Binding DeleteCommand}" RenderTransformOrigin="0.458,0.547"
              />
    </StackPanel>

    <TextBlock Name="txtBlockMessage"
           Grid.Row="6" Grid.ColumnSpan="4" Margin="12,15,50,10"
           Text="{Binding Path=Message}"
           />

    <DataGrid Name="dgEmployees"
          AutoGenerateColumns="False"
          Grid.Row="3"
          Grid.Column="1" Margin="38,10,13,10" Padding="3,3,3,3"
          ItemsSource="{Binding Path=EmployeesList,Mode=TwoWay}"
          RenderTransformOrigin="0.503,1.137" Grid.ColumnSpan="3" Grid.RowSpan="3">

      <DataGrid.Columns>
        <DataGridTextColumn Header="Employee Id"
                    Width="auto"
                    Binding="{Binding Path=Id}"/>

        <DataGridTextColumn Header="Employee Name"
                    Width="auto"
                    Binding="{Binding Path=Name}"/>

        <DataGridTextColumn Header="Employee Age"
                    Width="auto"
                    Binding="{Binding Path=Age}"/>
      </DataGrid.Columns>



    </DataGrid>

  </Grid>
</UserControl>
```

Fig. 7: Example code for *EmployeeView. XAML*

Description: Here, it bridges between view and model class. For example, we added only one ViewModel type. In an actual project, there are several classes inside the project folder. Couples of objects are created when it is instantiated. One is the *EmployeeService* object. It is responsible for operations like add, delete, search, etc. It also initiates a current employee class. It is used to save the data whenever the user input into the Inputbox. It is also used to pass the current employee object as a parameter to the methods. The relay commands like SaveCommand, search command, and update and delete command also initiate here. When the user presses the button inside the GUI, the program pointer will go to the RelayCommand object and then navigate here. It calls the ObjEmployee service function to process the task. It is also coordinated among various component modules. The EmployeeViewModel.cs is depicted in figures 5(a) to 5(c).

```
<Window x:Class="MVVM_Demo.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:MVVM_Demo"
      mc:Ignorable="d"
      Title="MainWindow" Height="700"
      Width="800"
      xmlns:vw="clr-namespace:MVVM_Demo.View">
   <Grid>
      <vw:EmployeeView Margin="0,10,10,-31"/>
   </Grid>
</Window>
```

Fig. 8: Example code for MainWindow. XAML

View Design: if we see the architecture of the MVVM according to the class dependency, The view is the last element in the actual scenario. Different teams design the view part, not in model or view model teams. It is the job of the UX design team. Inside the project folder, create a folder called "View." We add a user control, "*EmployeeView.XAML*". Copy-paste the code from figures 6 and figure 7. The view is where users input and display the data they need to see. If we observe, its text is bound with a variable in the coordinator class *EmployeeViewModel.cs*.

```
using MVVM_Demo.ViewModel;
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 MVVM_Demo
                                                                {
   public partial class MainWindow : Window
   {
      EmployeeViewModel ViewModel;

      public MainWindow()
      {
         InitializeComponent();
         ViewModel=new EmployeeViewModel();
         this.DataContext=ViewModel;
      }
   }
}
```

Fig. 9: Example code for MainWindow

Final Work: Now, only one job is pending. The gateway of the application is MainWindows. XAML. So we have to bind our view with these windows. Figure 8 depict the code which needs to add to the MainWindows.Xaml. And in the MainWindow. XAML.cs class add code from figure 9.
Execute: Now, build the code. It should successfully build. If not, we need to debug. Press the run button. The application Interface should look like 10. Now enter your id, name, and Age. Press add. The Employee will add to this list. After adding a couple of Employees, if we press the delete button, the Employee will delete where the id is in the id box. The search and update are self-explanatory. The application interface and GUI are for demonstration purposes only. We need to design GUI as per our requirement, but the procedure is the same as we described.

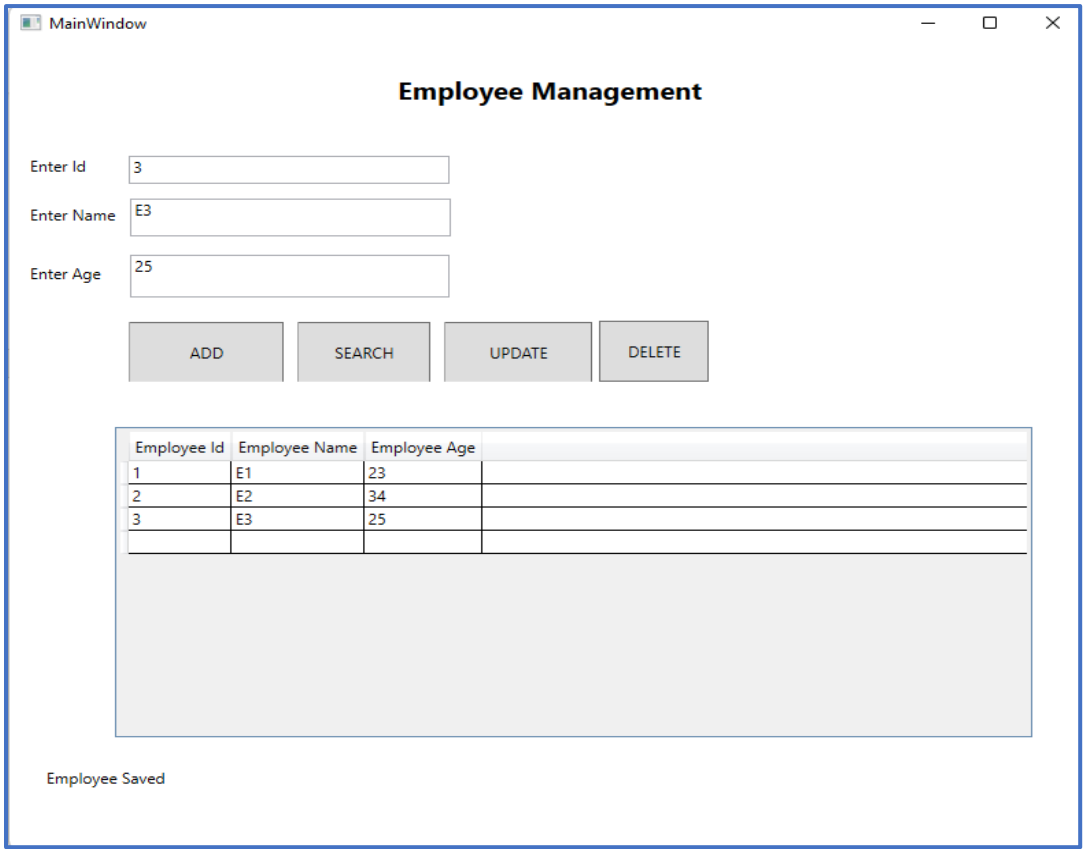

Fig. 10: Application Interface

6. RECOMMENDATIONS :

- For project code: https://github.com/sudipchakraborty/MVVM-Demonstration-Using-C-WPF.git
- We adopted the example code from: https://www.youtube.com/watch?v=rSrc252Bf3Y&list=PLEGjYQQO3ST8hatajWNDUGVwo437sPv8G&index=4
- SourceTree download link: https://www.sourcetreeapp.com/
- Good tutorial on C# MVVM: https://www.youtube.com/watch?v=i_DlDYFR0Ag&list=PLEGjYQQO3ST8hatajWNDUGVwo437sPv8G&index=1

7. CONCLUSION :

Preferrable software architecture is MVVM when we consider designing our software across a different team like UX design, programming, etc. It is loosely binding among various software components. That is why final integration among them is an excellent task. It is also ideal when we think about future maintainability. It is a language-independent architecture that can be applied to most standard languages. Here using C# and WPF, we implemented this design concept. This work might be a good reference for those trying hard to integrate this concept into their project.

REFERENCES :

[1] Pan, H. H., Jiang, J. J., Chen, L., Sun, H. T., & Tan, H. Q. (2011). A scalable graphical user interfaces architecture for CNC application based-On WPF and MVVM. In *Advanced Materials Research* (Vol. 317, pp. 1931-1935). Trans Tech Publications Ltd. Google Scholar↗

[2] Sorensen, E., & Mikailesc, M. (2010). Model-view-ViewModel (MVVM) design pattern using Windows Presentation Foundation (WPF) technology. *MegaByte Journal*, *9*(4), 1-19. Google Scholar↗

[3] Grossu, I. V., Opritescu, M., Savencu, O., Miron, A. I., Verga, M., & Verga, N. (2022). A new version of Hyper-Fractal Analysis: Net WPF module for RGB 3D reconstruction of medical three-channel images. *Computer Physics Communications*, *276*, 108335. Google Scholar↗

[4] Sulistyarini, D. D., Isman, R. K., & Maulana, H. (2018). Build and Design of Voyage Account Applications Using C#, WPF, and SQL Server 2012 (Case Study Company X). *INKOM Journal*, *11*(1), 25-32. Google Scholar↗

[5] Troelsen, A., & Japikse, P. (2017). WPF Notifications, Validations, Commands, and MVVM. In Pro C# 7 (pp. 1137-1176). Apress, Berkeley, CA. Google Scholar↗

[6] James, B., & Lalonde, L. (2015). Pro XAML with C#: Application Development Strategies (covers WPF, Windows 8.1, and Windows Phone 8.1). Apress. Google Scholar↗

[7] Sannarangaiah, K. (2020). Design and development of a graphical user interface with state-of-the-art C# patterns. Google Scholar↗

[8] James, B., & Lalonde, L. (2015). What Is XAML? In Pro XAML with C# (pp. 3-13). Apress, Berkeley, CA. Google Scholar↗

[9] Yuen, S. (2020). Mastering Windows Presentation Foundation: Build responsive UIs for desktop applications with WPF. Packt Publishing Ltd. Google Scholar↗

[10] Sheikh, W., & Sheikh, N. (2020). Audiometry: A model-view-ViewModel (MVVM) application framework for hearing impairment diagnosis. *Journal of Open Source Software*, *5*(51), 2016, 1-6. Google Scholar↗

****