`

Strato IT

디자인 패턴에 대한 연구

(MVC, MVP, MVVM)

팀장	신찬솔
팀원	김민철
	심종현
	이규성

목 차

1. 서론

1-1. 패턴이란?

사람은 모두 다르기 때문에 코딩 스타일부터 프로젝트 구조까지 전부 제각각이다. 이런 상황이 발생한다면 협업을 할 때 서로 같은 언어를 쓰더라도 상대방이 어떤 구조로 만들었고, 어떤 스타일로 코딩하는지 파악하는데 엄청난 시간이 소비될 것이다. 또한, 협업이 아니더라도 이전에 했던 코드가 어떤 일을 하는지 나중에 다시 보았을 때, 처음부터 구조를 파악하는 자신의 모습을 심심치 않게 볼 수 있을 것이다. 하지만 코드를 패턴화 함으로 써 한눈에 파악할 수 있는 구조를 만들어 시간 절약 및 가독성을 높일 수 있다

1-2. 디자인 패턴이란?

현재 프로그래밍에 통용되는 디자인 패턴은 여러가지이다. 우리는 많은 패턴 중 MVC, MVP, MVVM 패턴에 대해 알아볼 것이다. 이 3가지 패턴은 객체 지향 프로그래밍을 설계할 때 자주 발생하는 문제들을 피하기 위해 사용되는 패턴이고, 유지보수와 확장성 및 관리를 원활하게 하기 위해 사용하는 목적임을 인지해야 한다.

2. 본론

2-1 MVC 패턴

2-1-1 정의

MVC는 Model, View, Controller의 약자로서 소프트웨어 디자인 패턴이다.

객체를 역할별로 분리하여 효율적인 코드를 할 수 있도록 만들었다.

모델(Model)

내부적으로 사용되는 데이터를 저장, 처리하는 역할을 하는 비즈니스 로직이다.

데이터를 가공하는 곳이라고 칭한다(DB 접근, 추가, 수정 등 일을 처리)

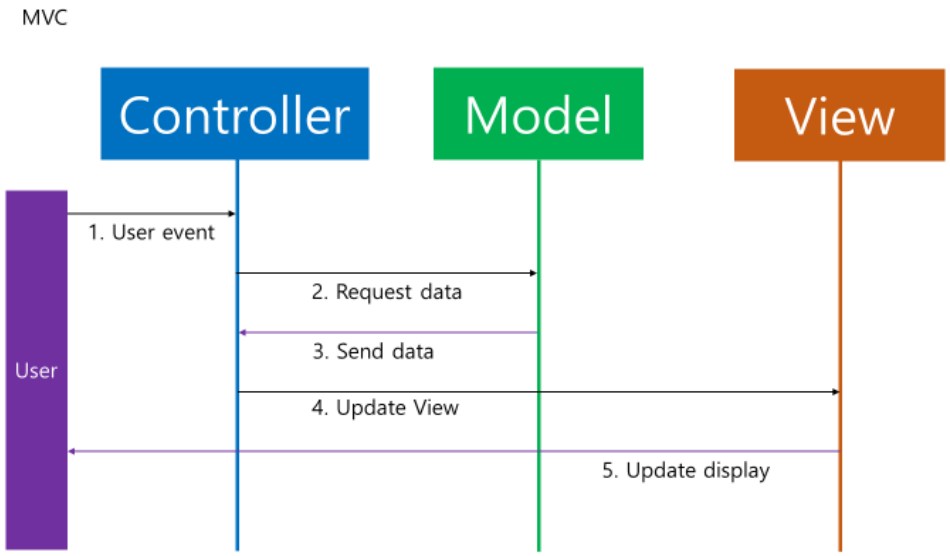
뷰(View)

주로 사용자에게 보여지는 인터페이스(UI) 영역을 뜻한다. Model에서 처리된 데이터를 받아서 사용자에게 보여주는 역할을 한다. Model과 종속 관계이다.

컨트롤러(Controller)

Model과 View의 Bridge 역할을 하기 때문에 Model이랑 View의 존재에 대해 알고 있어야 하며, 모델과 View의 변화를 감지하여 처리해주는 역할을 한다.

2-1-2 동작 순서(MVC)



[MVC - 동작원리]

1. User의 Action이 Controller에 Input

2. 관련 데이터를 Model에 요청

3. Model에서 관련 데이터를 Controller로 전송

4. Controller가 처리한 데이터를 View로 전송

5. View에서 User에게 화면을 출력해줌

User event를 controller에서 처리하고, event에 따른 data logic을 수행한다.

*** wpf에서 MVC패턴으로 나눈다면 Model, View(xaml), Controller(code-behind)로 나눌 수 있다.**

2-1-3 코드 설명(MVC)

MVC 패턴의 예시를 보여주기 위해 WPF로 구현한 MBStore_MVC 프로젝트를 사용한다.

```
namespace MBStore_MVC.Model
{
    public class Sign_up
    {
        public string Login_id{ get; set; }
        public string Login_pw{ get; set; }
        public string Gender{ get; set; }
        public string Social_number{ get; set; }
        public string Phone{ get; set; }
        public string Address{ get; set; }
        public string Sign_date{ get; set; }
        public string Email{ get; set; }
        public string Name{ get; set; }
        public string Post_number{ get; set; }
    }
}
```

<Model-property>

```
public List<Sign_up> GetList_Sign_Up_Emp()
{
    List<Sign_up> Signup_List = new List<Sign_up>();
    using (conn = new SqlConnection())
    {
        string sql = "select Login_id, Gender, Name, Social_number, Phone, " +
         "Email, Post_number, Address, Sign_date from signup";
        conn.ConnectionString =
           ConfigurationManager.ConnectionStrings["UserDB"].ToString();
           conn.Open();    //  데이터베이스 연결
        SqlCommand cmd = new SqlCommand(sql, conn);
        using (SqlDataReader myDataReader = cmd.ExecuteReader())
        {
             while (myDataReader.Read())
             {
                 Sign_up sign_up = new Sign_up();
                 sign_up.Login_id = myDataReader.GetString(0);
                 sign_up.Gender = myDataReader.GetString(1);
                 sign_up.Name = myDataReader.GetString(2);
                 sign_up.Social_number = myDataReader.GetString(3);
                 sign_up.Phone = myDataReader.GetString(4);
                 sign_up.Email = myDataReader.GetString(5);
                 sign_up.Post_number = myDataReader.GetString(6);
                 sign_up.Address = myDataReader.GetString(7);
                 sign_up.Sign_date = myDataReader.IsDBNull(8) ? "없음" :
                 myDataReader.GetDateTime(8).ToShortDateString();
                 Signup_List.Add(sign_up);
             }
        }
   }
   return Signup_List;
}
```

<Model – DB를 통해 값 얻기>

Comment

Visual studio 와 Mssql을 연결하기 위해 ado.net 방식을 사용했으며 GetList_Sign_Up_Emp함수 안에 SqlConnection을 사용하여 서버 연결을 담당하고 SqlCommand는 서버에서 실행될 sql 문을 전달한다. 원하는 값을 sql 문을 전달하여 값을 얻기 위해 데이터 조회를 담당하는 sqlDataReader를 이용하여 데이터를 얻는다. 우리는 더욱 쉽게 데이터를 얻기 위해 model에 property 객체를 만들어 List<>에 담아 쉽게 데이터를 저장할 수 있었다. 또한 using 문에 SqlConnection을 넣어 줌에 따라 값을 얻을 때만 DB에 접근하여 트래픽의 증가를 최소화 시켰다.

```
private void Btn_su_emp_signup(object sender, RoutedEventArgs e)
{
   lv_employee_sign_list.ItemsSource = null;
   lv_employee_sign_list.ItemsSource = db.GetList_Sign_Up_Emp();
}
```

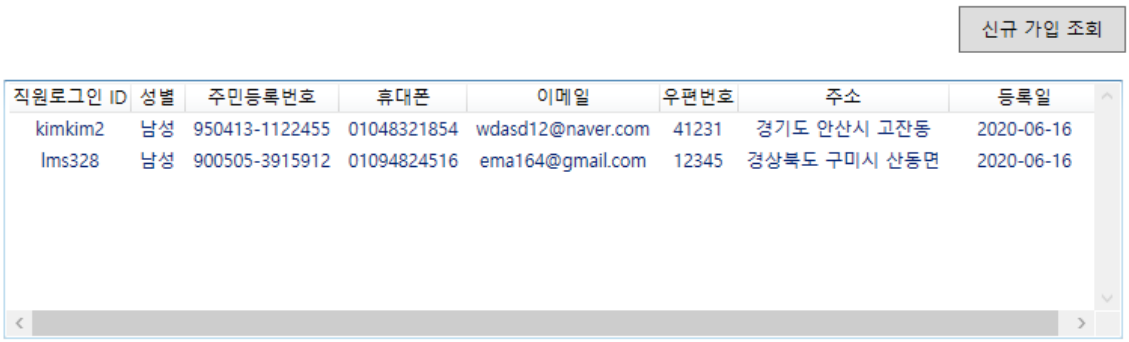
<Controller – Code Behind>

Comment

Controller 부분에서 Model과 View의 연결을 도와준다. View에 생성된 lv_employee_sign_list 안에 model에서 뽑아온 결과를 넣어주었다. 이로써 model과 view의 종속성 높아진다.

```
<Button x:Name="su_em_signup_search" Click="Btn_su_emp_signup" Grid.Row="10"
        Grid.Column="4" Margin="8" MinHeight="30" MinWidth="60">신규 가입 조회</Button>
<Button x:Name="su_em_signup_reset" Click="Btn_su_emp_signup_reset" Grid.Row="10"
        Grid.Column="5" Margin="8" MinHeight="30" MinWidth="60">초기화</Button>
<ListView Name="lv_employee_sign_list" Margin="10,10,10,10" Grid.Row="11"
        Grid.ColumnSpan="6" Grid.RowSpan="4">
        <ListView.View>
            <GridView>
                <GridViewColumn Width="80" Header="직원로그인 ID"
                      DisplayMemberBinding="{Binding Path=Login_id}"/>
                <GridViewColumn Width="60" Header="성별"
                      DisplayMemberBinding="{Binding Path=Gender}"/>
                <GridViewColumn Width="90" Header="주민등록번호"
                      DisplayMemberBinding="{Binding Path=Social_number}"/>
                 <GridViewColumn Width="80" Header="휴대폰"
                      DisplayMemberBinding="{Binding Path=Phone}"/>
                 <GridViewColumn Width="110" Header="이메일"
                      DisplayMemberBinding="{Binding Path=Email}"/>
                 <GridViewColumn Width="60" Header="우편번호"
                      DisplayMemberBinding="{Binding Path=Post_number}"/>
                 <GridViewColumn Width="130" Header="주소"
                      DisplayMemberBinding="{Binding Path=Address}"/>
                 <GridViewColumn Width="100" Header="등록일"
                      DisplayMemberBinding="{Binding Path=Sign_date}"/>
              </GridView>
        </ListView.View>
</ListView>
```

<View – XAML>



<View-UI>

Comment

View는 Xaml로 이용하여 UI를 구성하고 Controller에서 추출한 값을 DisplayNumberBinding을 통해 path에 Model에서 선언된 Property Name을 지정하면 DB에서 가져온 값이 자동으로 값이 할당된다.

2-1-4 장・단점(MVC)

<장점>

- 각각 패턴들을 구분하여 개발하므로 유지 보수가 용이하고 유연성과 확장성이 높다.(유지 보수 비용 절감)

- 1개의 Controller를 통해 View와 Model을 연결하므로 쉽게 기능 구현이 가능하다.

- 소규모 프로젝트에 적합하다.

<단점>

- model과 view의 의존성이 높아 어플리케이션이 커질수록 복잡하고 유지보수가 어렵다.

- Controller를 통해 View와 Model이 연결되어 프로젝트가 커진다면 수정 시 테스트가 힘들고 파악이 어렵다. (side-effect의 문제발생)

- 기본기능 설계를 위해 클래스들이 많이 필요하기 때문에 복잡하다.

2-2 MVP 패턴

2-2-1 정의

MVP 패턴은 Model, View, Presenter의 앞 글자를 따서 이름이 지어졌으며, MVC 패턴에서 파생된 디자인 패턴이다. 이 패턴의 핵심은 MVC 패턴의 단점인 View와 Model의 의존성을 해결하기 위하여 서로 간의 상호작용을 Presenter에 위임해 서로 영향을 미치게 하지 않는 것이다.

모델(Model)

내부적으로 사용되는 데이터를 저장, 처리하는 역할을 하는 비즈니스 로직이다. View, Presenter 모두에게 의존적이지 않은 독립적이다.

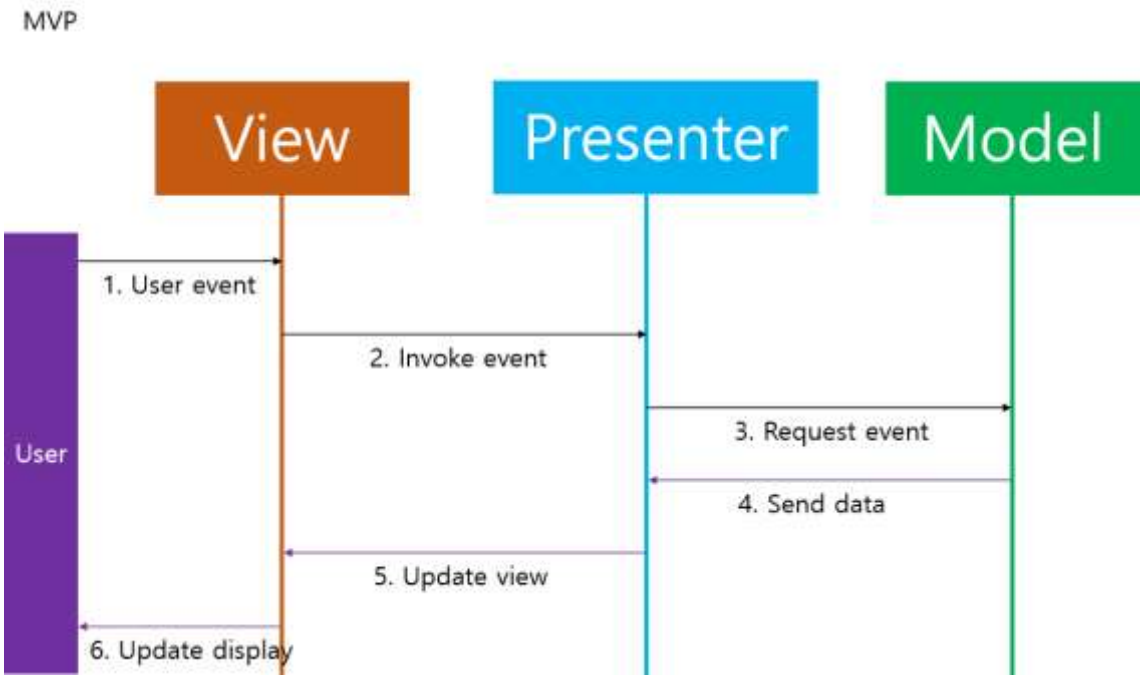
뷰(View)

주로 사용자에게 보이는 인터페이스(UI) 영역을 뜻한다. 사용자의 Action을 감지하여 Presenter에 보내주며, Model에서 처리된 데이터를 Presenter를 통해 받아서 사용자에게 보여주는 역할을 한다. Presenter에게 의존적이다.

프레젠터(Presenter)

Model과 View 사이의 매개체 역할을 하여 View에서 발생한 Action을 받아 Model에게 전달하고, Model의 로직에서 나온 결과를 다시 View에게 전달하는 하는 매개체 역할을 한다. Model과 View 모두에게 의존적이다.

2-2-2 동작 순서(MVP)



[MVP-동작원리]

1. User의 Action이 View로 Input

2. View는 데이터를 Presenter에게 요청

3. Presenter는 Model에게 데이터 요청

4. Model은 요청받은 데이터를 Presenter로 전송

5. Presenter는 View에게 데이터 전송

6. View에서 User에게 화면을 출력해 줌

*** 얼핏 보면 MVC 패턴과 차이가 없어 보일 수 있으나, MVC에선 View만을 위한 로직이 없고 출력만을 담당한다. 그리고 Presenter(Controller)에 대한 호출의 흐름이 MVC에선 View 안의 Action이 Controller를 호출한다는 점이 MVP와 차이점을 가진다.**

2-2-3 사용예시(MVP)

MVP 패턴의 예시를 보여주기 위해 WPF로 구현한 MBStore_MVP 프로젝트를 사용한다.

```
interface IMainWindow_view
{
        void emp_signupList_reset();
 }
```

<View의 Interface>

```
namespace MBStore_MVP.View
{
    public partial class MainWindow : Window, IMainWindow_view
    {
        IMainWindow presenter;

        public MainWindow() : base()
        {
            InitializeComponent();
            presenter = new Pre_MainWindow(this);
        }

        private void Btn_su_emp_signup(object sender, RoutedEventArgs e)
        {
            lv_employee_sign_list.ItemsSource = presenter.GetList_Sign_Up_Emp();
        }

        private void Btn_su_emp_signup_reset(object sender, RoutedEventArgs e)
        {
            presenter.emp_signup_reset();
        }

        public void emp_signupList_reset()
        {
            lv_employee_sign_list.ItemsSource = null;
        }
    }
}
```

<View – xaml.cs>

```
<Button Click="Btn_su_emp_signup" Grid.Row="10" Grid.Column="4" Margin="8"
        MinHeight="30" MinWidth="60">신규 가입 조회</Button>
<Button Click="Btn_su_emp_signup_reset" Grid.Row="10" Grid.Column="5" Margin="8"
        MinHeight="30" MinWidth="60">초기화</Button>
<ListView Name="lv_employee_sign_list" Margin="10,10,10,10" Grid.Row="11"
        Grid.ColumnSpan="6" Grid.RowSpan="4">
        <ListView.View>
            <GridView>
                <GridViewColumn Width="80" Header="직원로그인 ID"
                      DisplayMemberBinding="{Binding Path=Login_id}"/>
                <GridViewColumn Width="60" Header="성별"
                      DisplayMemberBinding="{Binding Path=Gender}"/>
                <GridViewColumn Width="90" Header="주민등록번호"
                      DisplayMemberBinding="{Binding Path=Social_number}"/>
                 <GridViewColumn Width="80" Header="휴대폰"
                      DisplayMemberBinding="{Binding Path=Phone}"/>
                 <GridViewColumn Width="110" Header="이메일"
                      DisplayMemberBinding="{Binding Path=Email}"/>
                 <GridViewColumn Width="60" Header="우편번호"
                      DisplayMemberBinding="{Binding Path=Post_number}"/>
                 <GridViewColumn Width="130" Header="주소"
                      DisplayMemberBinding="{Binding Path=Address}"/>
                 <GridViewColumn Width="100" Header="등록일"
                      DisplayMemberBinding="{Binding Path=Sign_date}"/>
              </GridView>
        </ListView.View>
</ListView>
```

<View – Xaml>

Comment

MVC에서는 Controller의 역할을 수행하던 code-behind는 MVP에서는 XAML과 함께 View의 역할을 가지게 된다. XAML 코드에서 이벤트를 발생할 컨트롤과 이벤트 결과를 담기 위한 컨트롤의 이름을 지정해 주고, code-behind에서 발생한 이벤트 처리하기 위한 함수를 생성해 준다. 그리고 Presenter에게 데이터 처리를 전달한다.

code-behind는 인터페이스 IMainWindow_view를 상속받아 구현한다. Presenter는 presenter의 부모 인터페이스 IMainWindow로 정의한 뒤 자식 클래스인 Pre_MainWindow로 할당하였다. 인터페이스를 사용하여 표준화를 제공받을 수 있고, 다형성, 클래스 간 결합도를 낮추었다.

<PRESENTER>

```
interface IMainWindow
{
        List<Sign_up> GetList_Sign_Up_Emp();

        void emp_signup_reset();
}
```

<Presenter의 인터페이스>

```
namespace MBStore_MVP.Presenter
{
    class Pre_MainWindow : IMainWindow
    {
        private readonly IMainWindow_view view;

        mbDB mbdb = new mbDB();

        public Pre_MainWindow(IMainWindow_view view)
        {
            this.view = view;
        }

        public List<Sign_up> GetList_Sign_Up_Emp()
        {
            return mbdb.GetList_Sign_Up_Emp();
        }

        public void emp_signup_reset()
        {
            view.emp_signupList_reset();
        }
     }
}
```

< Presenter >

Comment

View에서 받은 이벤트를 처리하기 위해 Model에게 데이터 처리를 요청 후 결과값을 받아 return 하여 View에게 전달해 준다.

인터페이스 IMainWindow를 상속받아 함수를 정의한다. View에 대해 직접적으로 접근하지 않고 View의 인터페이스를 통해 간접적으로 접근하여 결합도를 낮추었다.

<Model>

Presenter에서 받은 요청을 Model에서 데이터 처리 후 결과를 return 해준다

위의 MVC의 Model과 동일하다.

2-2-4 장・단점(MVP)

<장점>

- Model과 View간의 종속성를 낮춰준다.

- 새로운 기능을 추가하거나 수정이 필요할 때 관련된 부분만 수정하면 되기 때문에 확장성이 좋아진다.

- 테스트 코드를 작성하기 편리해진다.

<단점>

- View와 Presenter간의 의존성이 생긴다

- Interface를 지속적으로 작성함으로써 boilerplate 코드가 많아짐

- 소규모 프로젝트 혹은 소규모 팀에서는 부담이 될 수 있음

2-3 MVVM 패턴

2-3-1 정의

MVVM은 Model - View - View Model의 약자로서 이전의 MVP 패턴에서 Presenter를 View Model로 바꾼 디자인 패턴이다. 다른 패턴들과 마찬가지로 Model은 데이터를 가지고 있고, View는 화면을 보여주는 부분을 담당하고 있다. 그리고 Presenter를 대체하는 View Model은 View에서 발생하는 이벤트를 처리하는 부분이다. MVVM은 WPF에 최적화되어있으며 MVP 패턴과 비교했을 때 View에 대한 의존성이 존재하지 않는다.

모델(Model)

내부적으로 사용되는 데이터를 저장, 처리하는 역할을 하는 비즈니스 로직이다.

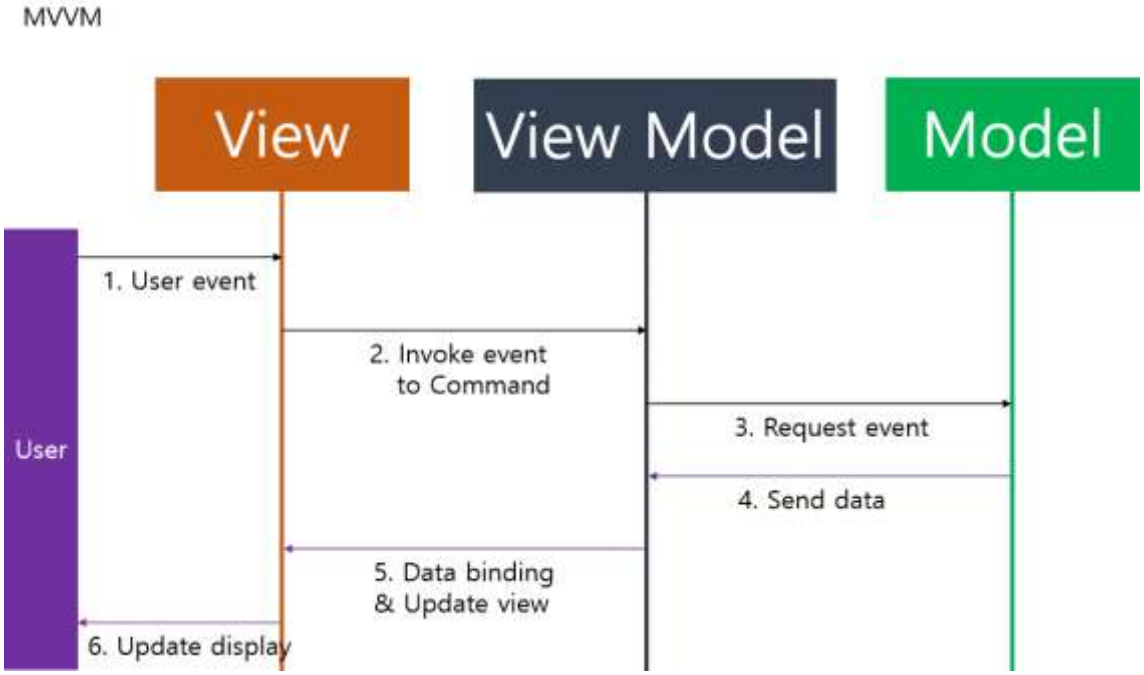
뷰(View)

주로 사용자에게 보여지는 인터페이스(UI) 영역이다. 사용자의 Action을 감지하여 ViewModel로 보내주며, Model에서 처리된 데이터를 ViewModel을 통해 받아서 사용자에게 보여주는 역할을 한다

뷰모델(View Model)

Model과 View 사이의 매개체 역할을 하고 기존의 MVP 패턴의 Presenter의 단점을 개선하였다. View에서 발생한 Action을 Command 패턴으로 받아 Model에게 전달하고, Model에서 나온 결과를 Data Binding하여 View에 전달하는 역할을 한다.

2-3-2 동작 순서(MVVM)



[MVVM-동작원리]

1. User의 Action이 View에 Input

2. Action이 들어오면 Command 패턴으로 View Model에 Action을 전송

3. View Model은 Model에게 데이터를 요청

4. Model은 View Model에게 요청받은 데이터를 전송

5. View Model은 받은 데이터를 가공 및 처리

6. View는 View Model과 Data Binding되어 사용자에게 화면을 출력

2-3-3 사용예시(MVVM)

실제 MVVM 패턴이 적용된 사례를 알아보기 위해 프로젝트로 구현한 MBStore_MVVM 프로젝트를 예제로 사용한다. MVVM 패턴을 적용하기에 앞서 필요한 사전 지식이 몇 가지 있다.

1. DataContext 연결
2. ICommand 인터페이스를 상속받는 클래스
3. 동작 방식에 따른 변화 감지

< DataContext >

MVVM 패턴의 구조상 View, ViewModel, Model은 각각 독립적이다. 이 말을 단순하게 생각하면 Code Behind가 비어있다는 말이 된다. 이렇게 서로 독립적인 각각이지만 데이터를 주고받으려면 최소한의 연결고리가 필요하다. 그 연결고리가 바로 DataContext이다. DataContext로 연결할 클래스 파일을 지정해주면 연결된 상태가 된다. 아래 소스코드를 예로 들면 ViewModel에서 new 로 View 객체를 생성하고 View의 Datacontext를 this로 ViewModel과 연결시켜준다. 그 아래의 소스코드는 View의 Code Behind이고 비어있는 것을 확인할 수 있다.

```
mainWindow_w = new MainWindow();
mainWindow_w.DataContext = this;
login_w.Close();
mainWindow_w.ShowDialog();
```

<login.xaml.cs>

```
public MainWindow()
{
    InitializeComponent();
}
```

<MainWindow.xaml.cs>

< 커맨드 기본 >

```
public class DelegateCommand : ICommand
{
    private readonly Func<bool> canExecute;
    private readonly Action execute;
    public DelegateCommand(Action execute) : this(execute, null) { }
    public DelegateCommand(Action execute, Func<bool> canExecute)
    {
        this.execute = execute;
        this.canExecute = canExecute;
    }
    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object o)
    {
        if (this.canExecute == null)
        {
            return true;
        }
        return this.canExecute();
    }
    public void Execute(object o)
    {
        this.execute();
    }
    public void RaiseCanExecuteChanged()
    {
        if (this.CanExecuteChanged != null)
        {
            this.CanExecuteChanged(this, EventArgs.Empty);
        }
    }
}
```

<View Model>

Comment

MVVM 패턴의 핵심기능 중 하나는 커맨드이다. 커맨드를 사용하기 위해서는 먼저 구현해야 하는 것이 바로 ICommand 인터페이스를 상속받는 클래스이다. 여기서는 DelegateCommand라는 이름으로 상속받는다.

ICommand를 상속받을 때는 반드시 Execute, CanExecute함수를 구현해주어야 한다. Execute함수는 함수 실행 시 이루어지는 작동을, CanExecute는 커맨드가 작동하기 위한 조건의 정의한다. 만약 무조건 실행하려면 CanExecute에 return true를 넣어주면 된다.

< INotifyPropertyChanged, ObservableCollection >

MVVM 패턴과 다른 패턴과의 차이점 중 하나는 Refresh 메서드가 없다는 것이다. ListView의 경우 MVC에는 항목의 추가 / 제거가 이루어지면 Refresh를 통해 목록을 갱신해 주어야 한다. 하지만 MVVM은 Binding과 INotifyPropertyChanged, ObservableCollection을 이용하여 항목의 변화가 감지되면 자동으로 갱신이 이루어진다.

```
private ObservableCollection<Product> lv_lo_reg_productlistItemsSource;
public ObservableCollection<Product> Lv_Lo_Reg_ProductListItemsSource
{
    get { return lv_lo_reg_productlistItemsSource; }
    set
    {
        lv_lo_reg_productlistItemsSource = value;
        OnPropertyChanged("Lv_Lo_Reg_ProductListItemsSource");
    }
}
private List<Sign_up> _Lv_Emp_Sign_ItemsSource;
public List<Sign_up> Lv_Emp_Sign_ItemsSource
{
    get { return _Lv_Emp_Sign_ItemsSource; }
    set
    {
        _Lv_Emp_Sign_ItemsSource = value;
        OnPropertyChanged("Lv_Emp_Sign_ItemsSource");
    }
}
```

<View Model>

Comment

View와 프로퍼티를 연결하려면 ViewModel에서 프로퍼티를 정의해야 한다. 이때, OnPropertyChanged를 사용해야 한다. " "안의 내용이 바로 View에서 바인딩 된 프로퍼티의 이름이다. 상단의 프로퍼티는 자료형이 ObservableCollection<T>로 지정되어 있다. ObservableCollection<T>는 항목의 추가 / 삭제 등을 감지하는 기능을 한다. 하단의 List<T> 프로퍼티는 INotifyPropertyChanged인데 이는 값의 변화를 감지하고 변화를 자동으로 갱신하는 기능을 한다.

< Model – ObservableCollection<T> >

```
public ObservableCollection<Product> Get_Lo_Reg_RegistProductList()
{
    ObservableCollection<Product> productList = new ObservableCollection<Product>();
    try
    {
        using (conn = new SqlConnection())
        {
            conn.ConnectionString =
                ConfigurationManager.ConnectionStrings["userDB"].ToString();
            conn.Open();    //  데이터베이스 연결
            string sql = "select product_id, name, manufacture, cpu, inch, mAh, ram,
                brand, camera, weight, price, display, memory from product order by
                product_id asc";
            SqlCommand cmd = new SqlCommand(sql, conn);
            using (SqlDataReader reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    Product product = new Product();
                    product.Product_id = reader.GetInt32(0);
                    product.Name = reader.GetString(1);
                    product.Manufacture = reader.GetDateTime(2);
                    product.Cpu = reader.GetString(3);
                    product.Inch = reader.GetString(4);
                    product.MAh = reader.GetInt32(5);
                    product.Ram = reader.GetInt32(6);
                    product.Brand = reader.GetString(7);
                    product.Camera = reader.GetInt32(8);
                    product.Weight = reader.GetInt32(9);
                    product.Price = reader.GetInt64(10);
                    product.Display = reader.GetString(11);
                    product.Memory = reader.GetInt32(12);
                    productList.Add(product);
                }
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
    return productList;
}
```

<Model>

Comment

위 소스코드는 ObservableCollection<T> 를 적용한 Model 부분의 함수이다. MVVM이 아닌 다른 패턴의 경우는 Refresh 메서드가 있기 때문에 List<T> 를 이용해도 기능에는 영향이 없다. 그 외에 함수의 비즈니스 로직은 다른 패턴과 변함이 없다.

< 커맨드 사용 >

커맨드를 사용하기 위한 준비가 되었다.

```
<Button Command="{Binding Sign_Up_Command}" Grid.Row="10" Grid.Column="5" Margin="8"
MinHeight="30" MinWidth="60" Content="신규 가입 조회"/>

private ICommand _Sign_Up_Command;
public ICommand Sign_Up_Command
{
    get { return (this._Sign_Up_Command) ?? (this._Sign_Up_Command = new
DelegateCommand(Sign_Up_Print_All)); }
}
void Sign_Up_Print_All()
{
    try
    {
        List<Sign_up> sign_up_list = new List<Sign_up>();
        sign_up_list = db.GetList_Sign_Up_Emp();
        Lv_Emp_Sign_ItemsSource = sign_up_list;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex + " ");
    }
}
```

<커맨드 사용 방식(Xaml, View Model)>

Comment

커맨드를 사용하는 방법 역시 다른 프로퍼티들과 비슷하다. 먼저 XAML에서 커맨드의 이름을 바인딩 한다. 이름을 설정하는데 제약은 없다. 그리고 ViewModel에서 해당 커맨드를 구현한다. Get을 통해 DelegateCommand를 호출하고 원하는 기능을 만들어놓은 함수를 인자로 넣어주면 된다. ViewModel에 선언하기 때문에 View에 있는 컨트롤에 직접 접근하지 않고 위에서 선언한 프로퍼티를 통해 처리 과정을 진행한다. Model 부분은 이전 패턴들과 동일하게 사용할 수 있다.

```
private void Btn_su_emp_signup(object sender, RoutedEventArgs e)
// 지원 -> 직원관리 -> 회원등록 조회버튼
{
    List<Sign_up> emp_signup;
    lv_employee_sign_list.ItemsSource = null;
    emp_signup = db.GetList_Sign_Up_Emp();
    lv_employee_sign_list.ItemsSource = emp_signup;
}
```

<MVC, MVP 이벤트 선언 방식>

```
private ICommand _Sign_Up_Command;
public ICommand Sign_Up_Command
{
    get { return (this._Sign_Up_Command) ??
        (this._Sign_Up_Command = new DelegateCommand(Sign_Up_Print_All)); }
}
void Sign_Up_Print_All()
{
    try
    {
        List<Sign_up> sign_up_list = new List<Sign_up>();
        sign_up_list = db.GetList_Sign_Up_Emp();
        Lv_Emp_Sign_ItemsSource = sign_up_list;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex + " ");
    }
}
```

<MVVM 이벤트 선언 방식>

Comment

위의 두 소스코드는 동일한 기능을 하지만 패턴에 따라 구현하는 방법이 달라지는 것을 보여주는 예시이다. 위는 MVC패턴, 아래는 MVVM패턴이다. MVC 패턴은 Code Behind에서 이벤트처리기를 이용하기 때문에 View에 있는 요소들에 직접 접근이 가능하다. 하지만 MVVM은 직접 접근이 불가능하기 때문에 선언한 프로퍼티들을 이용하여 데이터를 처리한다. 얼핏 보기에도 MVVM이 복잡해보이지만 종속성을 없앨 수 있다는 장점이 있다.

< ViewModel에 이벤트만들기 >

단순한 버튼 클릭 이벤트 같은 것은 커맨드로 대체하는 것이 수월하지만 이벤트를 최대한 간략하게 줄이려 해도 커맨드로는 한계에 도달하는 순간이 찾아온다. 그래서 사용자가 원하는 이벤트를 Code Behind에 만들지 않고 ViewModel에 만들 수 있는 방법이 필요하다.

```
<i:Interaction.Triggers>
    <i:EventTrigger EventName="MouseDoubleClick">
        <i:InvokeCommandAction Command="{Binding Lv_Emp_DoubleClickCommand}"/>
    </i:EventTrigger>
</i:Interaction.Triggers>
```

<Xaml>

```
private ICommand _Lv_Emp_DoubleClickCommand;
public ICommand Lv_Emp_DoubleClickCommand
{
    get { return (this._Lv_Emp_DoubleClickCommand) ??
        (this._Lv_Emp_DoubleClickCommand = new DelegateCommand(Update_DB_New_Emp)); }
}
void Update_DB_New_Emp()
{
    try
    {
        emp_sign_up_Window = new Emp_Sign_up();
        emp_sign_up_Window.DataContext = this;
        emp_sign_up_Window.Show();
    }
    catch (Exception ex)
    {
    }
}
```

<View Model>

Comment

앞의 XMAL코드는 View에서 EventTrigger를 이용하여 더블클릭 이벤트를 지정하고 Command로 ViewModel과 연결시킨 것이다. 그리고 아래 코드처럼 ViewModel에서 다른 커맨드와 같이 선언 후 동작을 지정해주면 이벤트를 만들 수 있다. 하지만 이는 Visual Studio에서 자체적으로 지원하지 않아 NuGet패키지에 있는 System.Windows.Interactivity를 찾아 설치하고 참조추가해야 사용이 가능하다.

2-3-4 장・단점(MVVM)

<장점>

- 기존에 MVP와 비교했을 때, View가 단독적으로 이뤄져있기 때문에 개발자와 디자이너의 협업이 용이하다.

- EventHandler, Business Logic을 반복구현하는 번거로움을 줄인다.

- 대규모 프로젝트에 적합하다.

<단점>

- 다른 디자인 패턴들(MVC, MVP)에 비해 설계가 어렵다.

- 소규모 프로젝트에서는 오히려 다른 패턴이 더 효율적이다.

- 소규모 프로젝트에서는 Logic이 Main Contents보다 더 길어진다.

- 이벤트처리기를 선언하는 방식이 복잡해진다.

3. 결론 및 고찰

위 3가지 패턴에 대한 학습을 진행한 후 WPF를 사용하여 '매장관리 시스템'을 MVC, MVP, MVVM 각각의 패턴으로 구현하면서 프로젝트를 진행했다. 이번 프로젝트를 통해 각 패턴에 대한 장·단점을 직접 경험해보고 각 패턴별 특징에 대한 이해도를 높일 수 있었다. 또 앞으로 다른 프로젝트를 진행하면서 어떤 패턴이 적합한지 빠르게 판단할 수 있는 능력을 기를 수 있었고, 능동적으로 패턴을 적용해 효과적으로 개발할 수 있는 능력을 기를 수 있었다. 마지막으로 상황에 맞춰 알맞은 디자인 패턴을 사용한다면 시간 절약 및 유지 보수, 재사용성을 높여 효율적인 프로젝트를 개발할 수 있을 것이다.