MVVM-АРХІТЕКТУРА ДОДАТКІВ З ГРАФІЧНИМ ІНТЕРФЕЙСОМ

Лекція 03
Інструментальні засоби візуального програмування
ЧДБК, 2020

Питання лекції

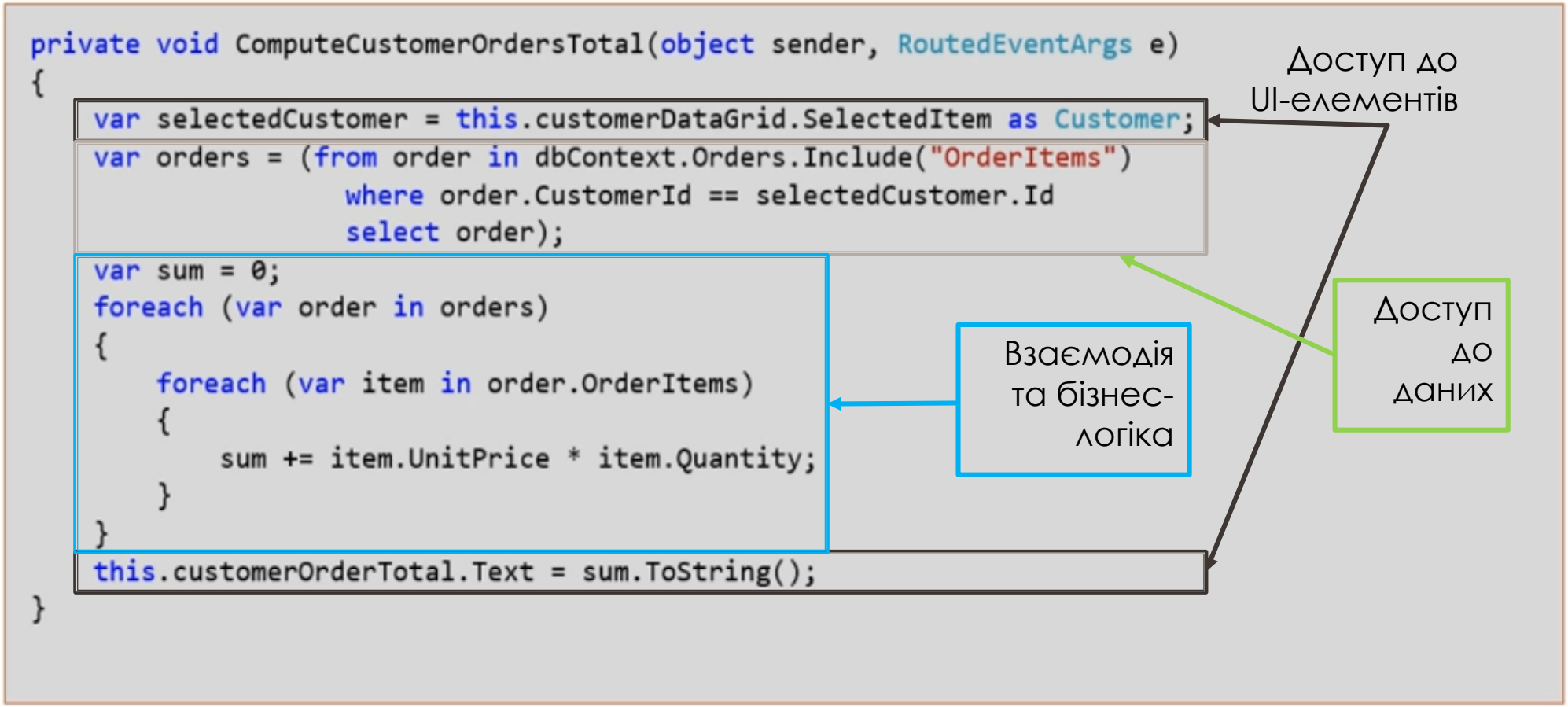
- Принципи роботи архітектури MVVM
- Формування рівня моделі та взаємодія з ним
- Поняття ресурсів у WPF-додатках
- Технологія прив'язування даних
- Стилізація елементів управління

ПРИНЦИПИ РОБОТИ АРХІТЕКТУРИ MVVM

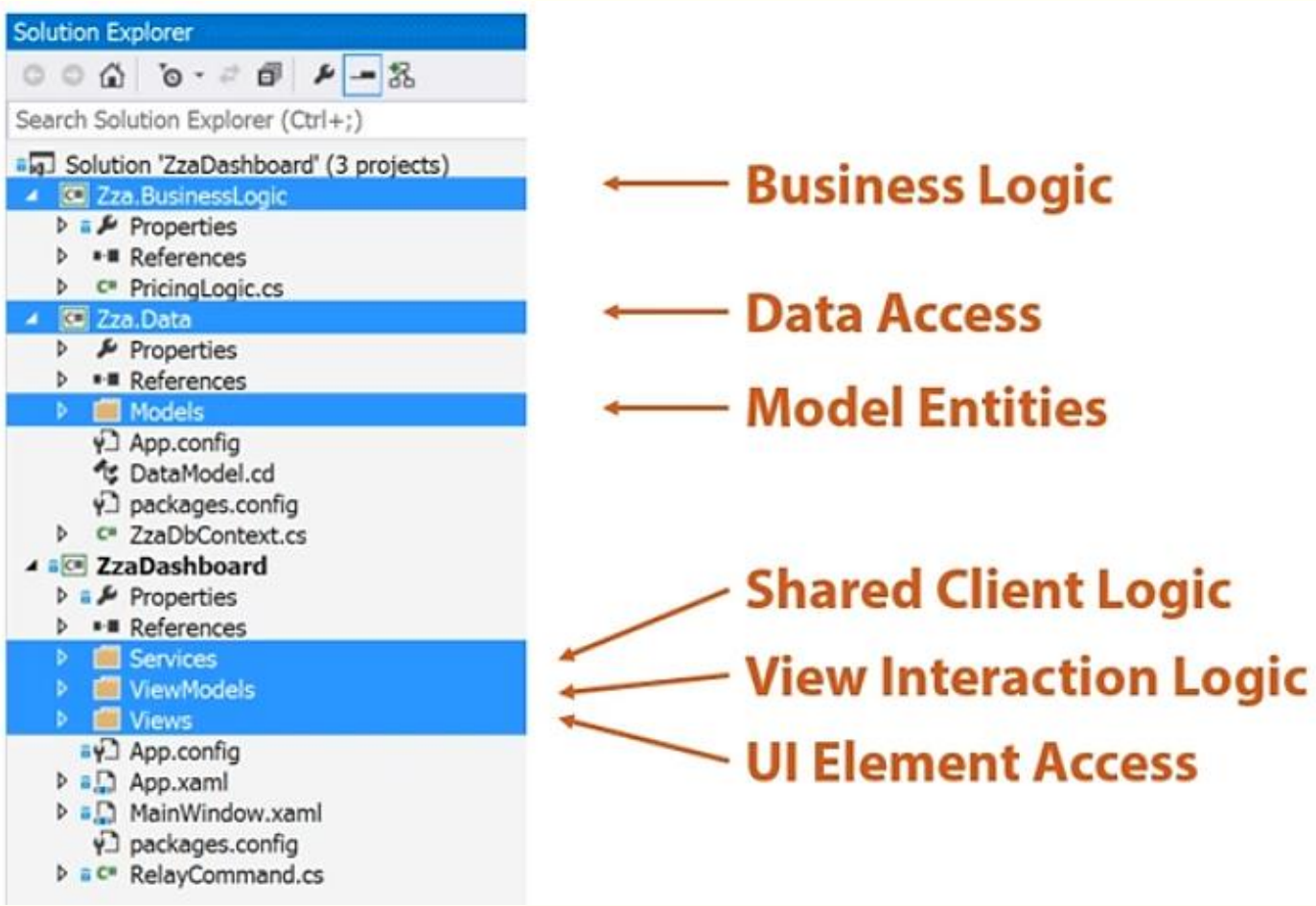
Питання 3.1.

Шаблон MVVM утворює розподіл відповідальностей (Separation of Concerns)

- Приклад коду без розподілу відповідальностей

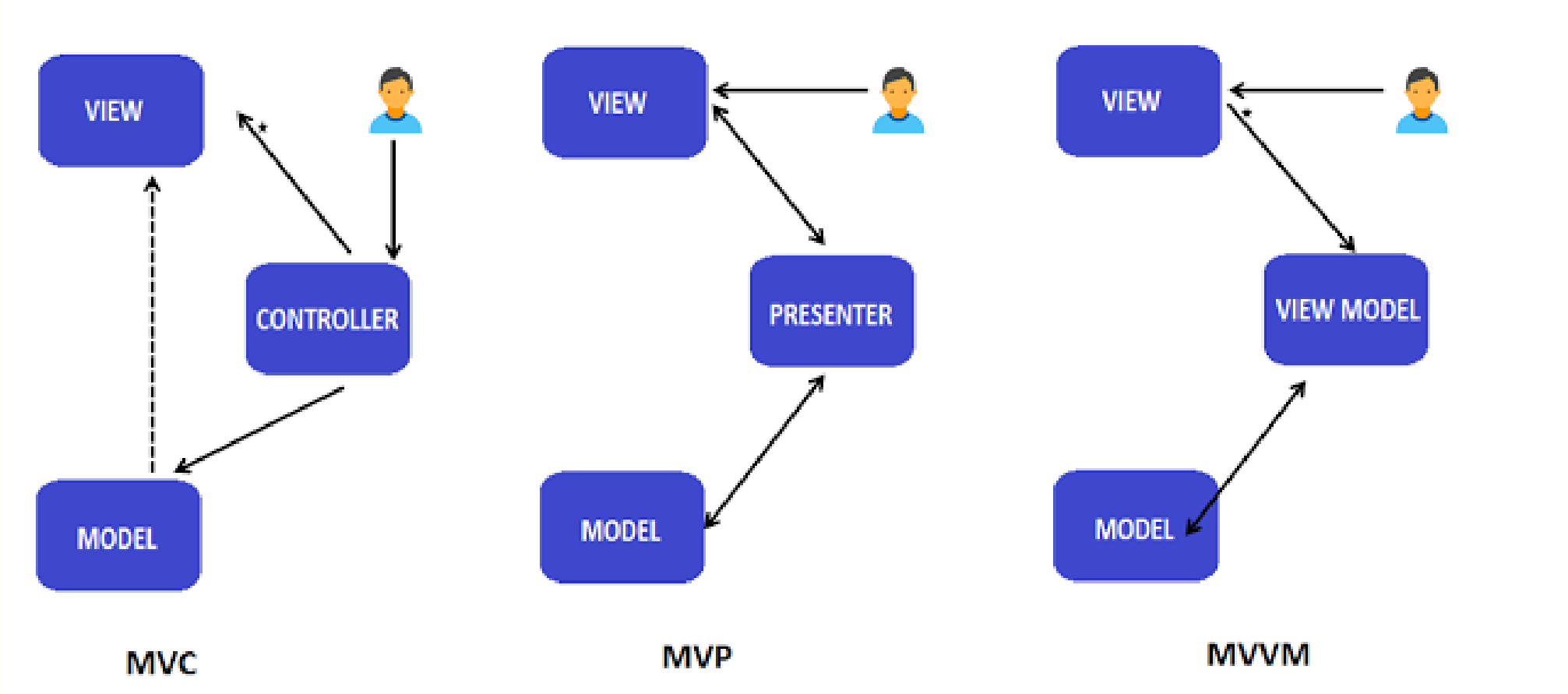


Розкладемо код «по поличках»



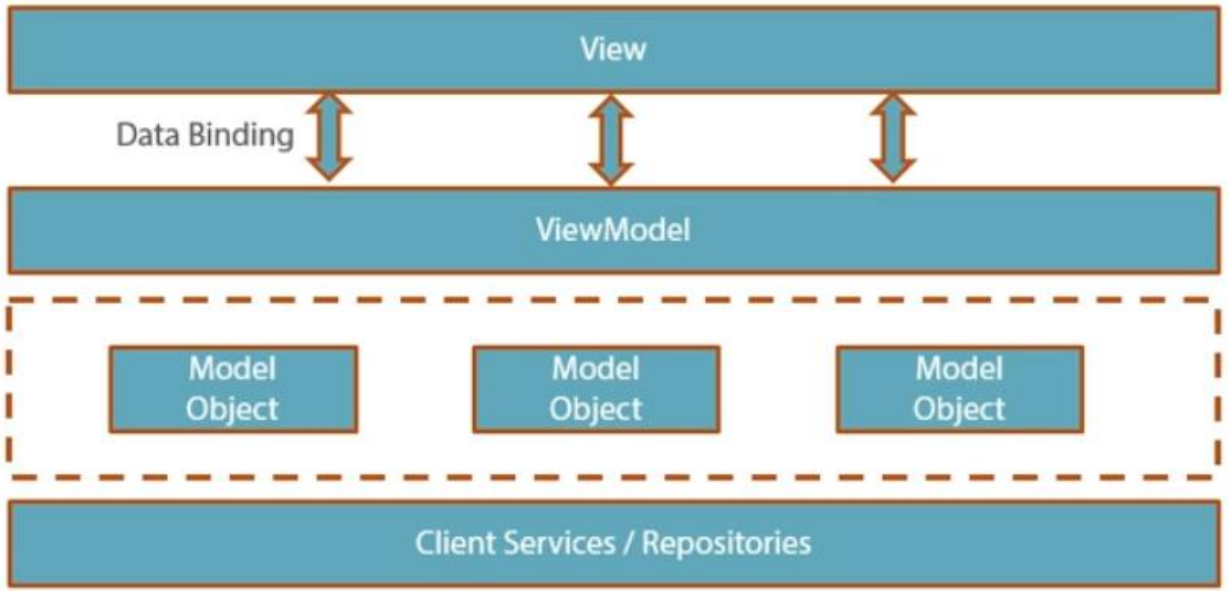
- Код простіше підтримувати
- Краща тестованість
- Краща розширюва-ність коду (extensibility)

MVVM як результат еволюції



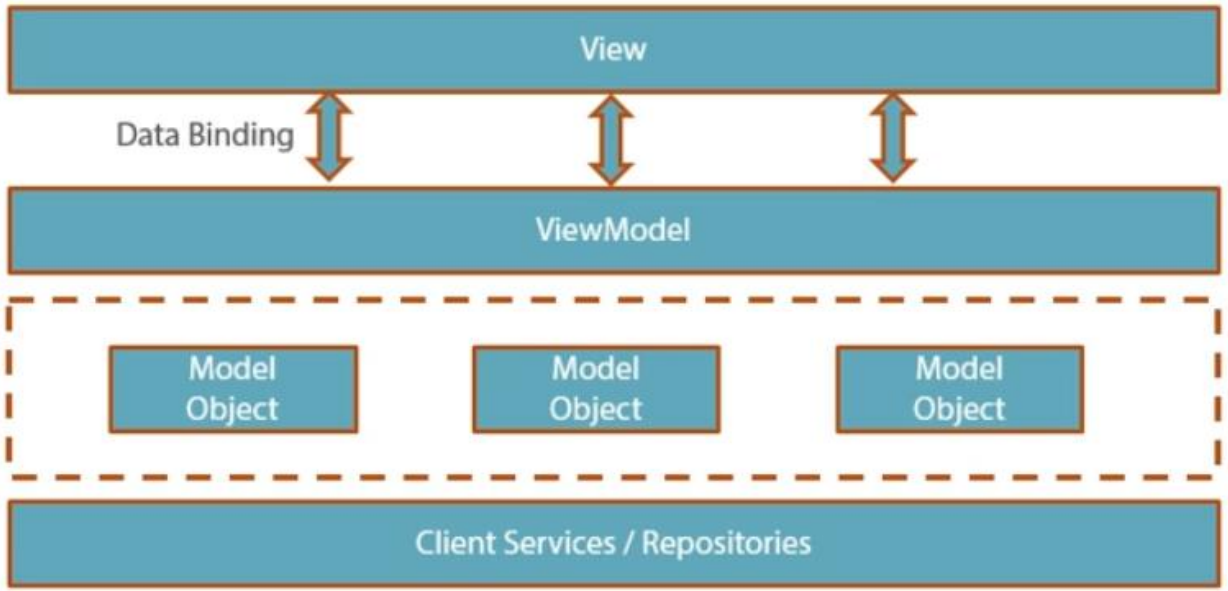
- https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad

Шаблон MVVM у WPF



- ***Рівень моделі*** описує клієнтську модель даних.
  - Складається з об'єктів з властивостями, у яких зберігаються окремі частини інформації.
  - Деякі властивості можуть розкривати зв'язки між об'єктами моделі – граф об'єктів (object graph).
  - Сюди відносять обчислювальні (computed) властивості, які отримують дані від інших властивостей або на базі інформації з контексту виконання додатку.
  - Інколи на рівень моделі виноситься валідація даних – у WPF це інтерфейси на зразок INotifyDataErrorInfo / IDataErrorInfo.
- ***Рівень представлення (View)*** займається структуруванням того, що користувач бачить на екрані. Структура може складатись зі
  - Статичних частин – XAML-ієрархія, яка визначає елементи управління та їх макетування
  - Динамічних частин – анімацій, змін стану.

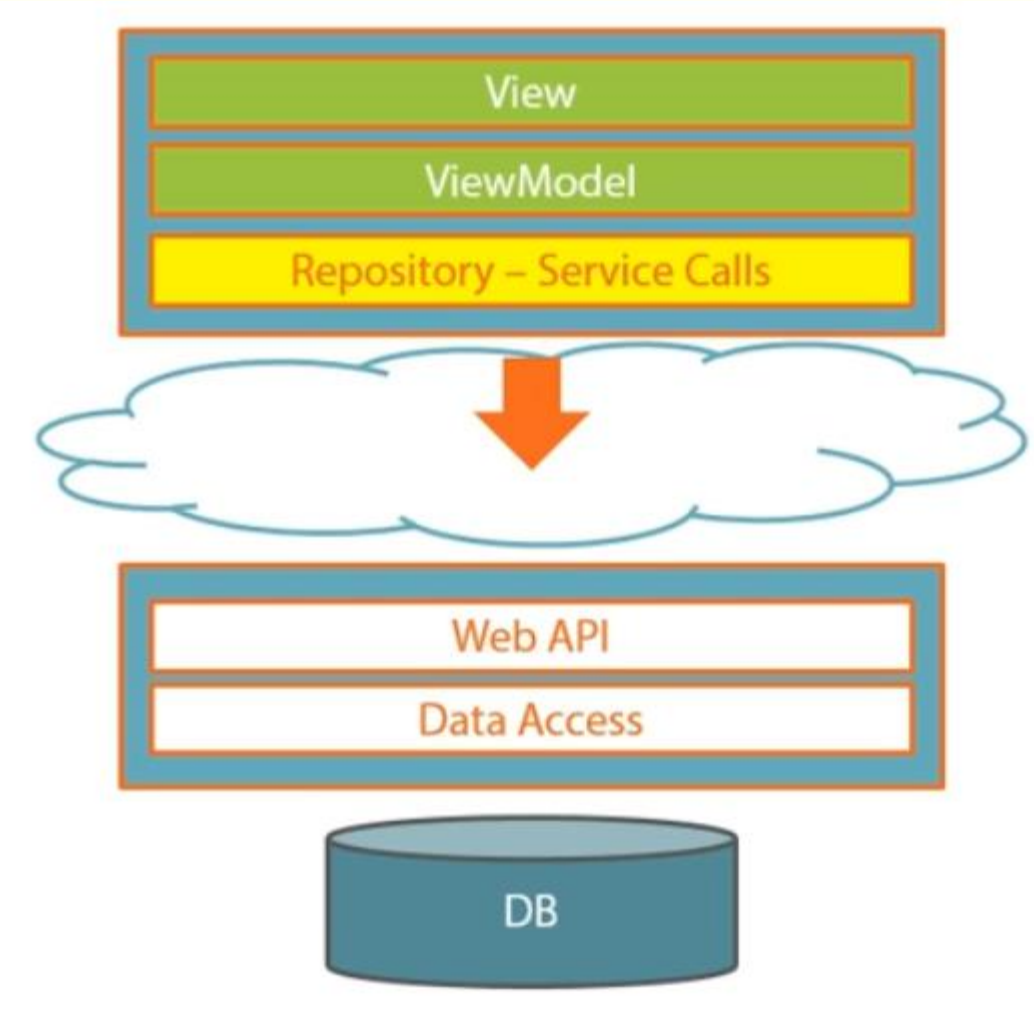
Шаблон MVVM у WPF



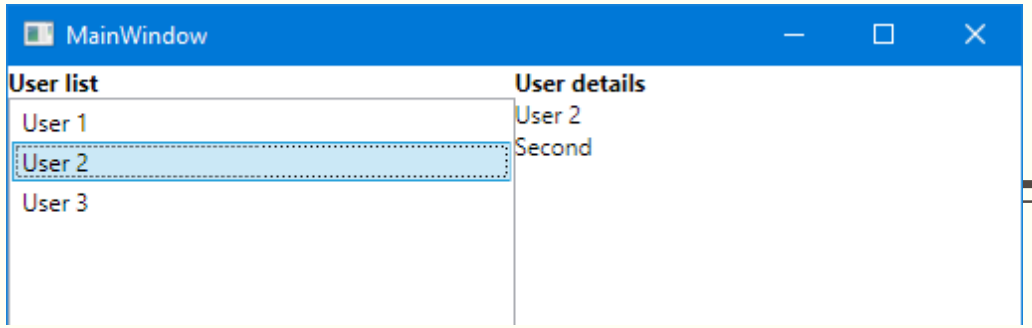
- ***Рівень ПредставленняМодель (ViewModel).***
  - Основна відповідальність – забезпечити рівень представлення даними та, за потреби, дозволити користувачу взаємодіяти з ними.
  - Ще одна ціль – інкапсулювати логіку взаємодії:
    - Звернення до бізнес-логіки/рівня даних/служб
    - Логіку навігації
    - Логіку перетворення станів.
- Фундаментальне рівняння MVVM:
  - **View.DataContext = ViewModel**

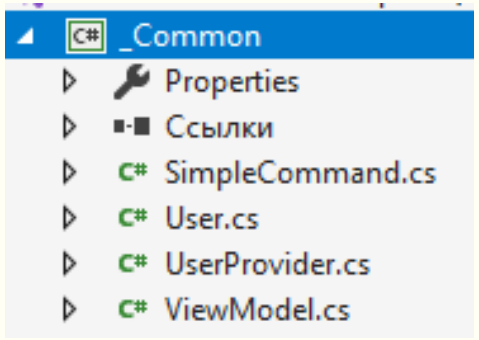
Рівень клієнтських служб (не є частиною офіційного MVVM)

- Інкапсулює будь-яку спільну логіку, яка розповсюджується на кілька ViewModel-ів.
- Код утиліт, доступу до даних, служби для підтримки безпеки тощо.
  - дозволяє абстрагуватись від речей, що можуть змінитись з часом.
    - наприклад, стратегії доступу до даних чи іншої функціональності, потрібної в кількох ViewModel.
- Способи побудови додатків:
  - ***View-First:*** спочатку конструюється представлення, потім – ViewModel, далі – прикріплення ViewModel-а до DataContext представлення.
  - ***ViewModel-First:*** спочатку конструюється ViewModel, потім - View як результат додавання ViewModel-а до інтерфейсу користувача



Демонстраційний додаток (спільні файли)

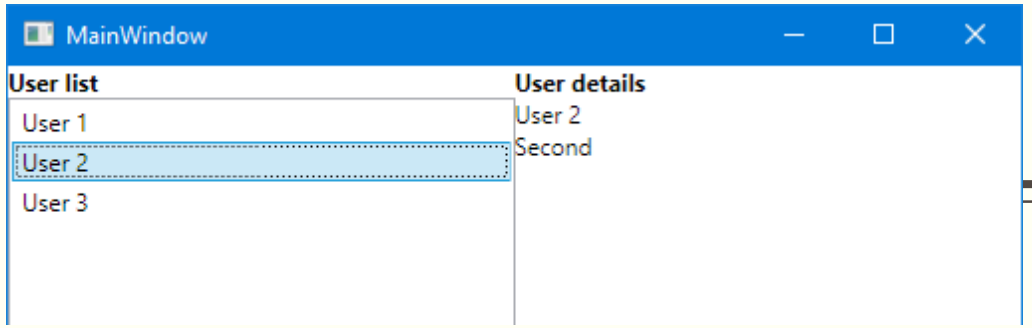





```
namespace Common
{
    public class User
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

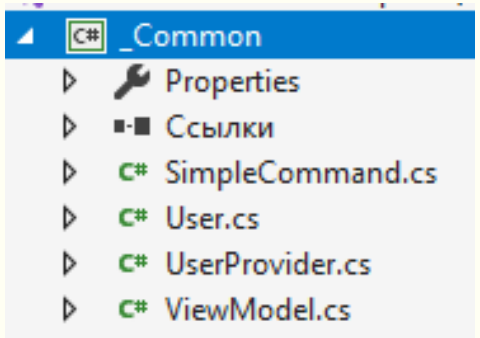
```
namespace Common
{
    public class UserProvider
    {
        public List<User> GetUsers()
        {
            return new List<User>
            {
                new User
                {
                    FirstName = "User 1",
                    LastName = "First"
                },

                new User
                {
                    FirstName = "User 2",
                    LastName = "Second"
                },

                new User
                {
                    FirstName = "User 3",
                    LastName = "Third"
                },
            };
        }
    }
}
```

- ***Рівень моделі містить предметну область додатку***

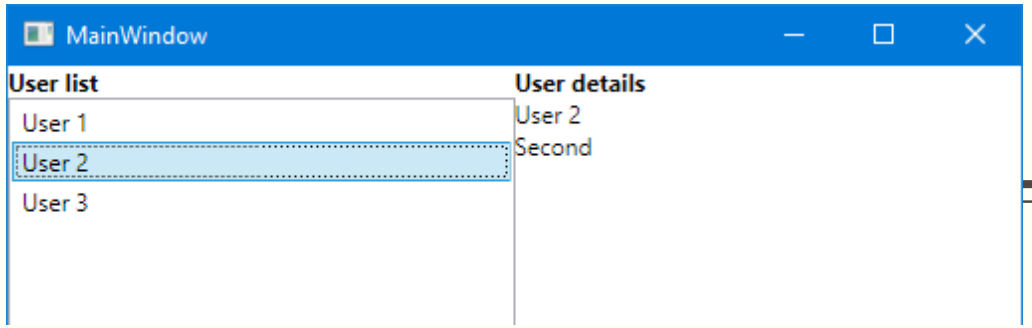


Демонстраційний додаток (спільні файли)

- *Частина рівня ViewModel, яка описує сповіщення про зміну значення властивості*

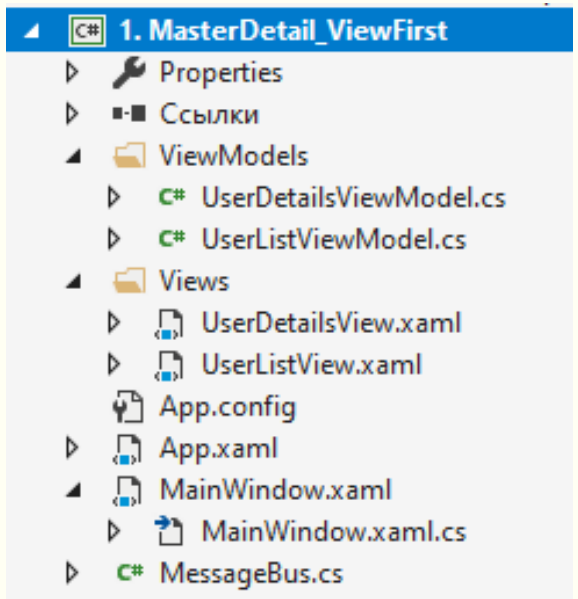



```
namespace Common
{
    public abstract class ViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```



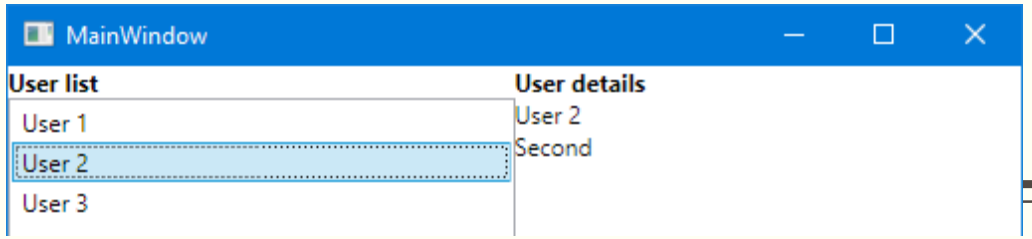
Демонстраційний додаток для підходу View-First

інтерфейс Master-Details



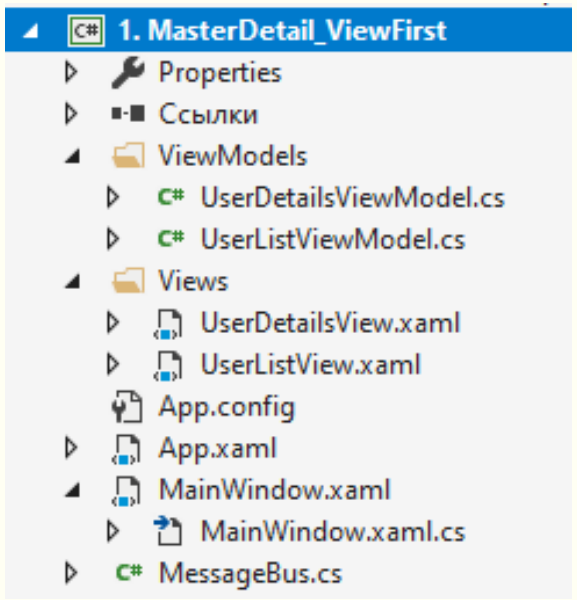
- *1. Створити Представлення (MainWindow.xaml)*

```
<Grid>

    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <view:UserListView Grid.Column="0" />

    <view:UserDetailsView Grid.Column="1" />

</Grid>
```



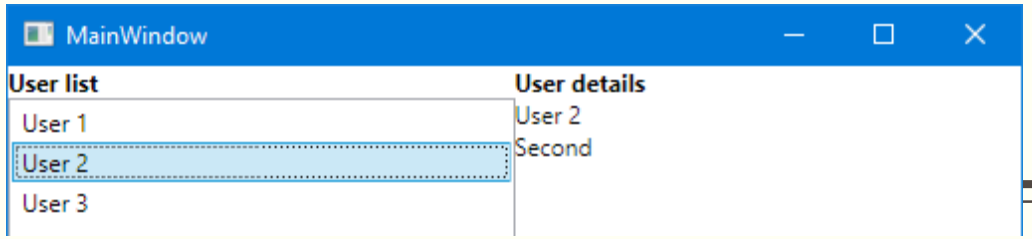
Демонстраційний додаток: частини інтерфейсу

- *Представлення (UserListView.xaml)*

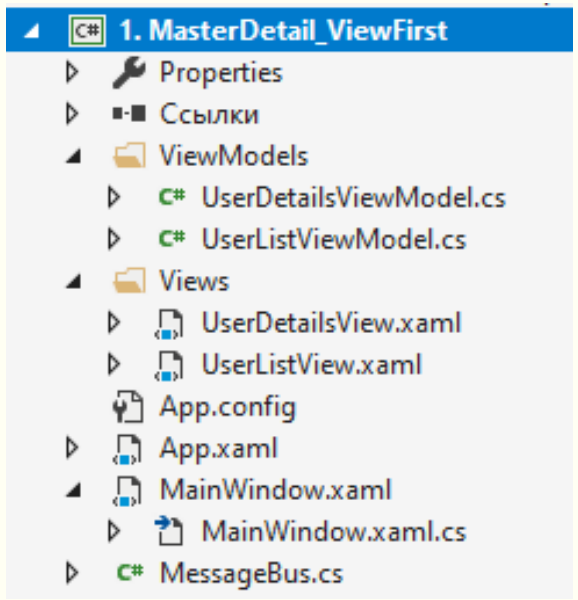



```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <TextBlock Grid.Row="0" Text="User list" FontWeight="Bold" />
        <ListBox Grid.Row="1" x:Name="UserListBox"
             SelectedItem="{Binding SelectedUser, Mode=TwoWay}"
             ItemsSource="{Binding Users}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding FirstName}" />
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
</Grid>
```

```
public partial class UserListView : UserControl {
        public UserListView() {
            InitializeComponent();
            DataContext = new UserListViewModel();
        }
    }
```



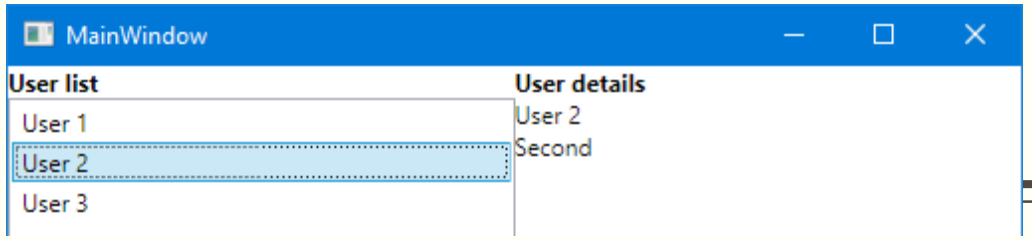
Демонстраційний додаток: частини інтерфейсу



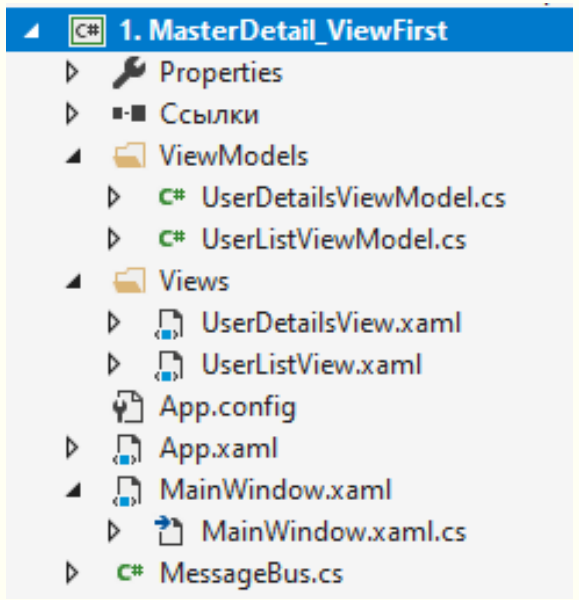
- *Представлення (UserDetailsView.xaml)*

```
    <StackPanel Orientation="Vertical">

        <TextBlock Text="User details" FontWeight="Bold" />

        <TextBlock Text="{Binding User.FirstName}" />
        <TextBlock Text="{Binding User.LastName}" />
    </StackPanel>
```

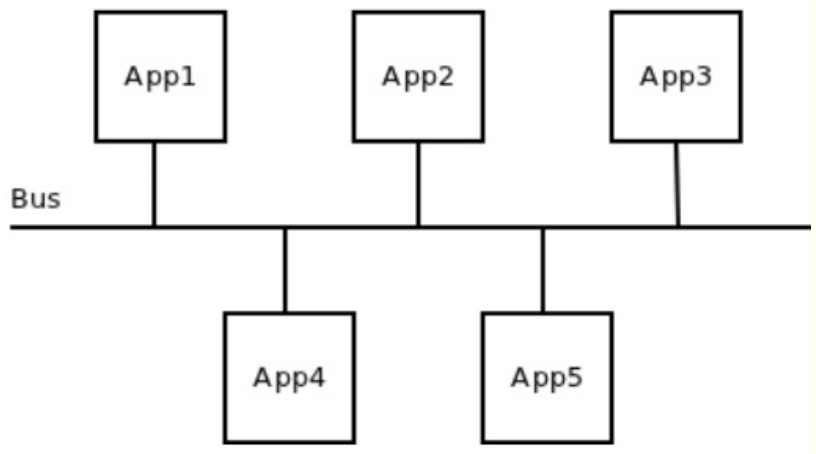
```
public partial class UserDetailsView : UserControl {
        public UserDetailsView() {
            InitializeComponent();
            DataContext = new UserDetailsViewModel();
        }
}
```



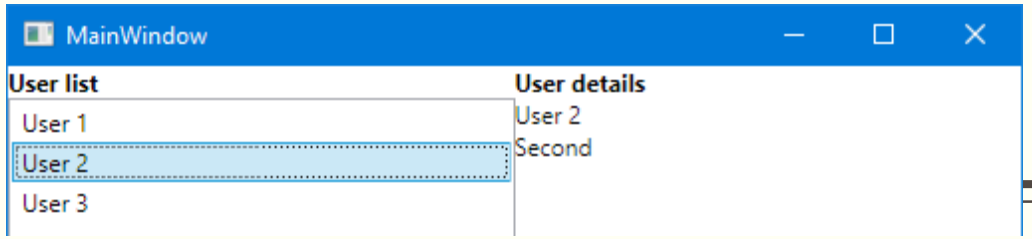
Демонстраційний додаток: шаблон MessageBus



- ***Для кожного представлення буде свій ViewModel-код***
  - Для синхронізації роботи представлень використовується шаблон MessageBus

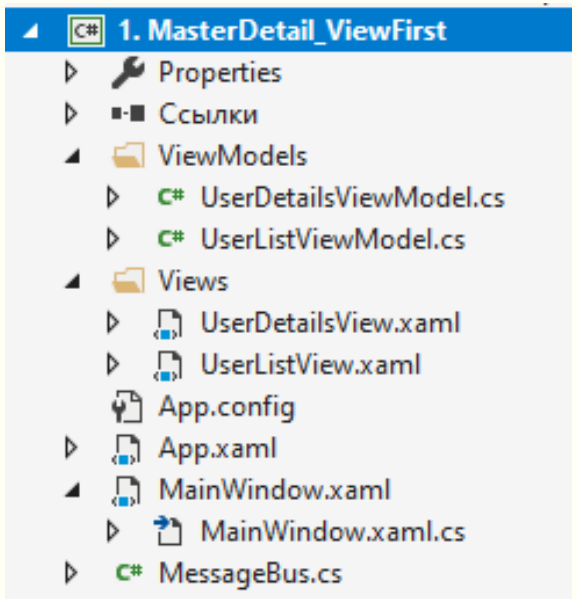


  - Шаблон важковаговий, використовується для інтеграції незалежних програмних компонентів, які нічого один про одного не знають.
  - Базується на використанні широкомовних подій.

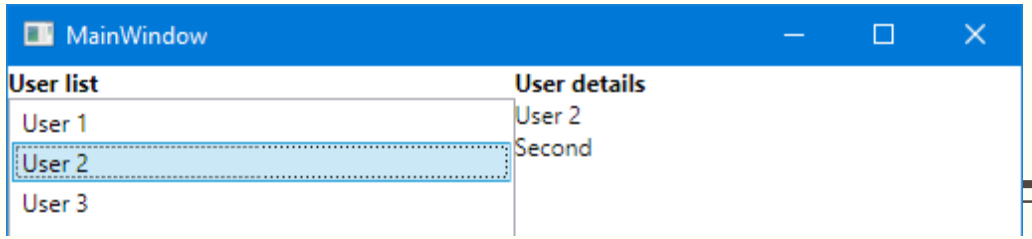


Демонстраційний додаток: шаблон MessageBus

- *Проста реалізація в даному контексті: для події SelectedUserChanged викликається UserChangedEventArgs з даними для синхронізації*

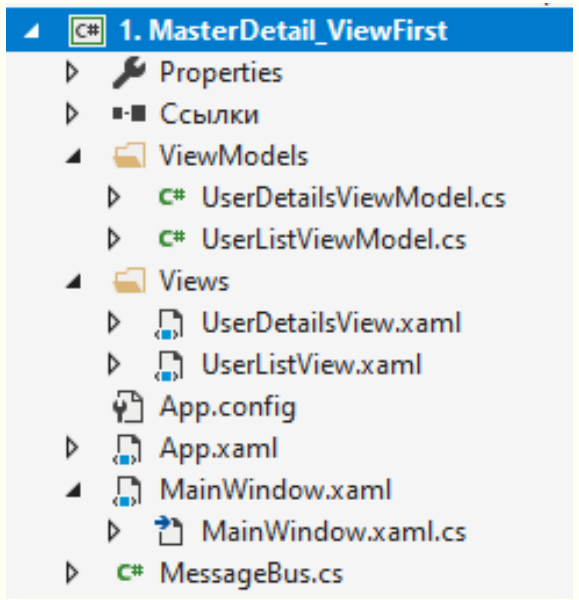



```
public class MessageBus {
    public static MessageBus Instance = new MessageBus();
    public event EventHandler<UserChangedEventArgs> SelectedUserChanged;

    public void OnSelectedUserChanged(User user) {
        SelectedUserChanged?.Invoke(this, new UserChangedEventArgs(user));
    }
}

public class UserChangedEventArgs : EventArgs {
    public UserChangedEventArgs(User user) {
        User = user;
    }
    public User User { get; }
}
```

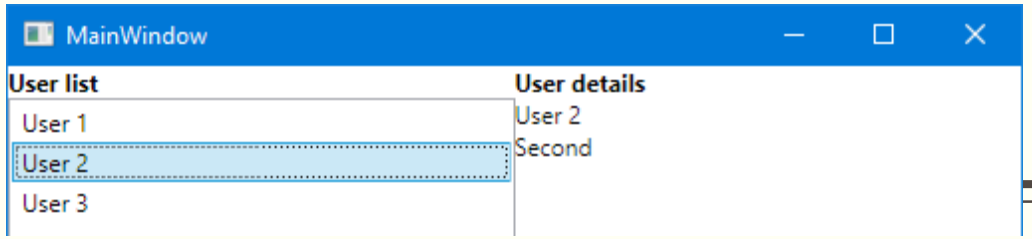


Демонстраційний додаток: *рівень ViewModel (UserListViewModel.xaml)*

- Додаємо використання MessageBus та кидаємо широкомовну подію про зміну користувача

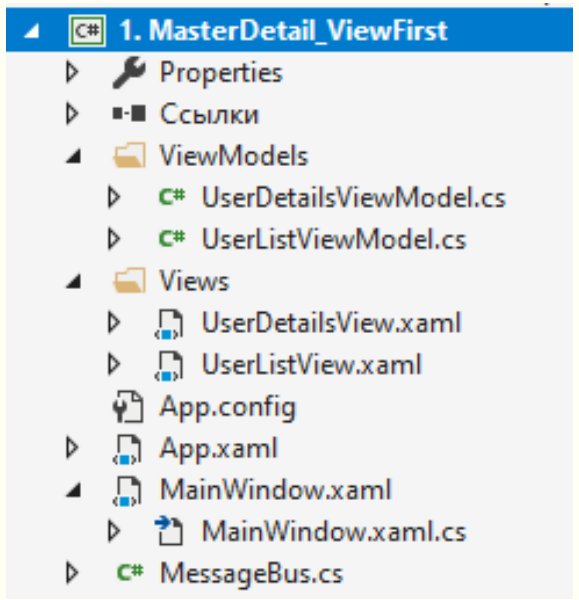



```
public class UserListViewModel : ViewModel {
        private IEnumerable<User> _users;
        private User _selectedUser;

        public IEnumerable<User> Users {
            get { return _users ?? (_users = new UserProvider().GetUsers()); }
        }

        public User SelectedUser {
            get { return _selectedUser; }
            set {
                _selectedUser = value;
                OnPropertyChanged();

                MessageBus.Instance.OnSelectedUserChanged(value);
            }
        }
    }
```



Демонстраційний додаток: *рівень ViewModel (UserDetailsViewModel.xaml)*

- Підписуємось на подію зміни користувача

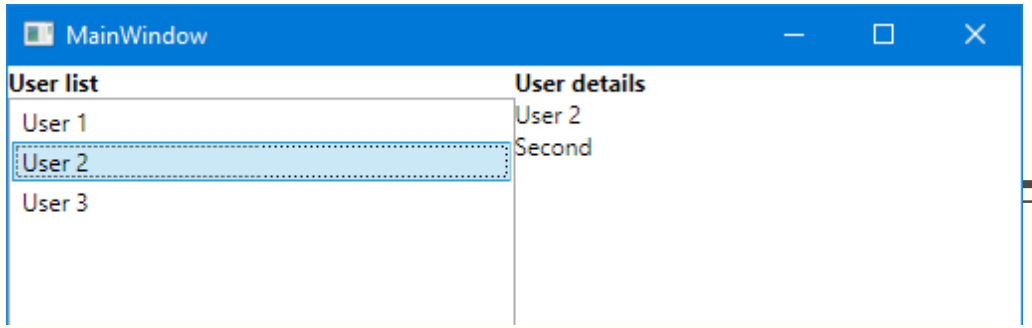



```
public class UserDetailsViewModel : ViewModel {
        public UserDetailsViewModel() {
            MessageBus.Instance.SelectedUserChanged += OnSelectedUserChanged;
        }

        private void OnSelectedUserChanged(object sender,
                                           UserChangedEventArgs userChangedEventArgs) {
            User = userChangedEventArgs.User;
        }

        private User _user;

        public User User {
            get { return _user; }
            set {
                _user = value;
                OnPropertyChanged();
            }
        }
}
```

Недоліки підходу

- Використовується важковаговий шаблон проектування, призначений для системної інтеграції
- На широкомовну подію може підписуватись будь-який об’єкт
- Поведінка системи стає заплутаною та неочевидною через покладання на широкомовні події
- Ускладнюється навігація по додатку (створення дочірніх форм)
  - Проектування навігації аналогічне веб-підходу:
  - Navigation.Show<ViewModel>(Value);
  - або
  - Navigation.Show(“View”, Value);

Демонстраційний додаток для підходу ViewModel-First

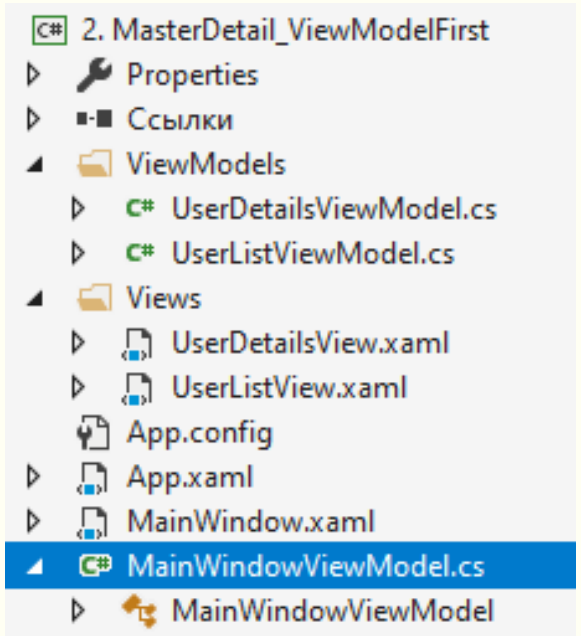


- *Повернемось до питання синхронізації форм:*
  - *Приберемо створення ViewModel у code-behind*

```
public UserListView()
{
    DataContext = new UserListViewModel();
}
public UserDetailsView()
{
    DataContext = new UserDetailsViewModel();
}
```

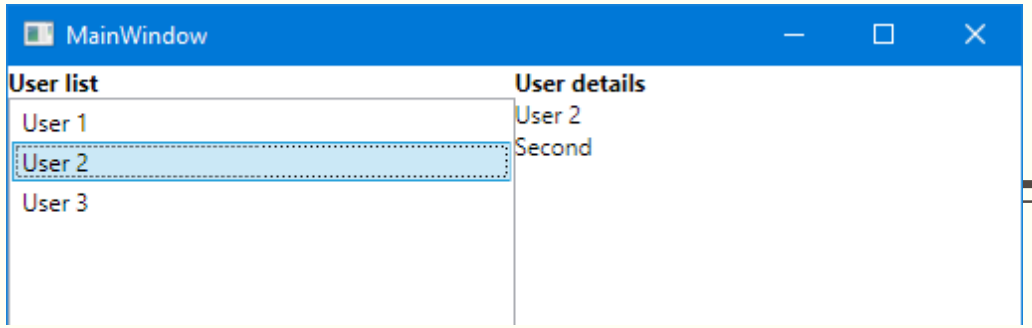
  - *Приберемо використання паттерну MessageBus*

інтерфейс Master-Details



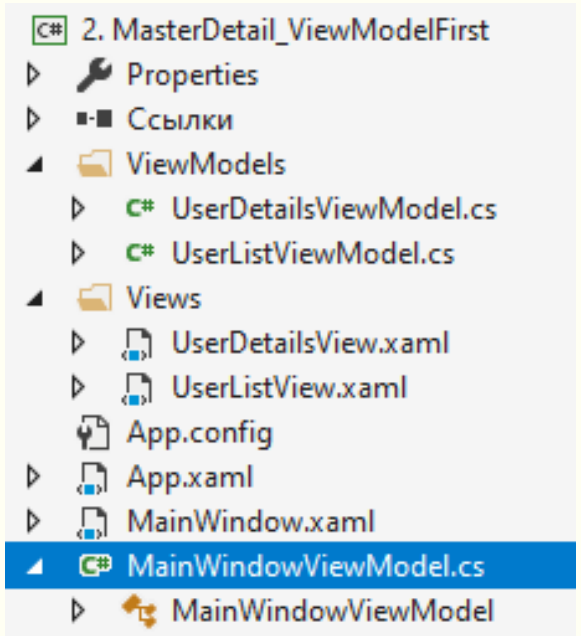

```
public class UserListViewModel : ViewModel
{
    public User SelectedUser
    {
        get { return _selectedUser; }
        set
        {
            _selectedUser = value;
            OnPropertyChanged();

            MessageBus.Instance.OnSelectedUserChanged(value);
        }
    }
}
```

```
public class UserDetailsViewModel : ViewModel
{
    public UserDetailsViewModel()
    {
        MessageBus.Instance.SelectedUserChanged +=
                (s, e) => User = e.User;
    }
}
```

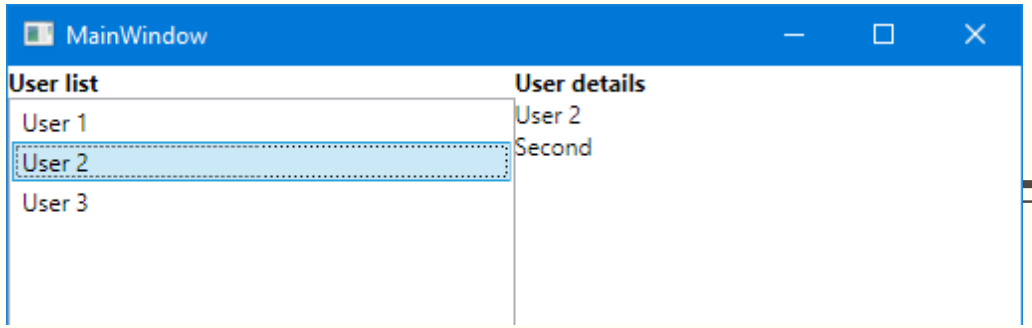


Демонстраційний додаток для підходу ViewModel-First

- *Синхронізація у стилі ViewModel-First – створення батьківського ViewModel-а.*
  - *Пряма, а не широкомовна підписка на події*

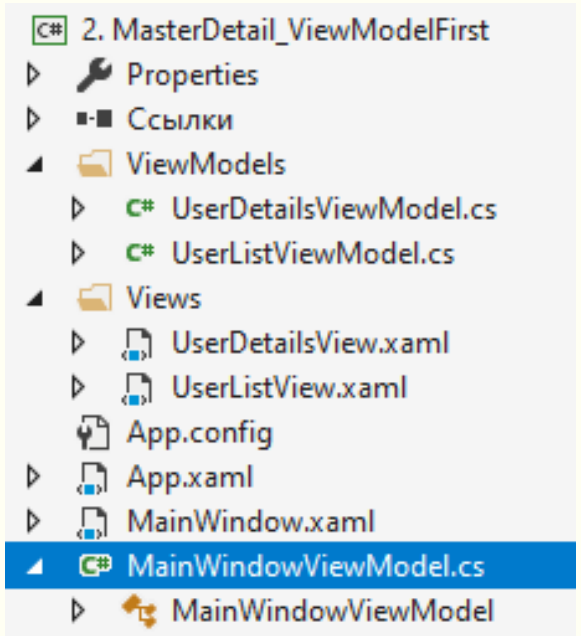



```
public class MainWindowViewModel : ViewModel {
        public UserDetailsViewModel UserDetailsViewModel { get; private set; }
        public UserListViewModel UserListViewModel { get; private set; }

        public void Initialize() {
            UserListViewModel = new UserListViewModel();
            UserDetailsViewModel = new UserDetailsViewModel();

            UserListViewModel.PropertyChanged += UserListViewModelOnPropertyChanged;
        }

        private void UserListViewModelOnPropertyChanged(object sender,
            PropertyChangedEventArgs propertyChangedEventArgs) {

            if (propertyChangedEventArgs.PropertyName == "SelectedUser") {
                UserDetailsViewModel.User = UserListViewModel.SelectedUser;
            }
        }
}
```



Демонстраційний додаток для підходу ViewModel-First

- *DataContext ініціалізується дещо по-іншому, на рівні представлення з батьківського ViewModel-а*




```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <views:UserListView
            DataContext="{Binding UserListViewModel}"
            Grid.Column="0" />

        <views:UserDetailsView
            DataContext="{Binding UserDetailsViewModel}"
            Grid.Column="1" />
    </Grid>
```

Інші приклади застосування підходів

GitHub

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Формування рівня моделі та взаємодія з ним