Архітектурний шаблон MVVM

Питання 4.3.

Шаблон MVVM

- Приклад коду без розподілу відповідальностей (Separation of Concerns)

```
private void ComputeCustomerOrdersTotal(object sender, RoutedEventArgs e)
{
    var selectedCustomer = this.customerDataGrid.SelectedItem as Customer;
    var orders = (from order in dbContext.Orders.Include("OrderItems")
                  where order.CustomerId == selectedCustomer.Id
                  select order);
    var sum = 0;
    foreach (var order in orders)
    {
        foreach (var item in order.OrderItems)
        {
            sum += item.UnitPrice * item.Quantity;
        }
    }
    this.customerOrderTotal.Text = sum.ToString();
}
```

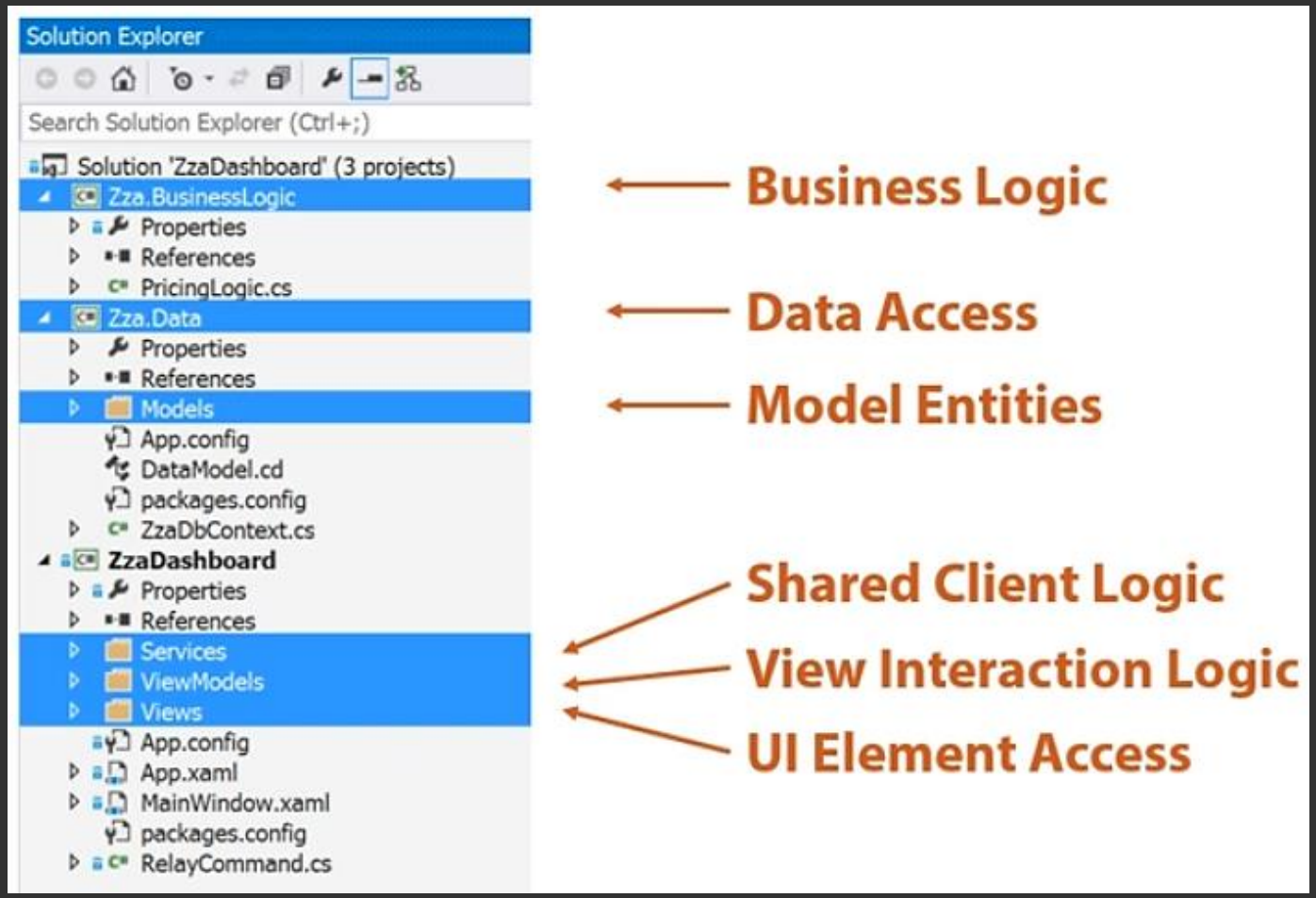
Доступ до UI-елементів

Взаємодія та бізнес-логіка

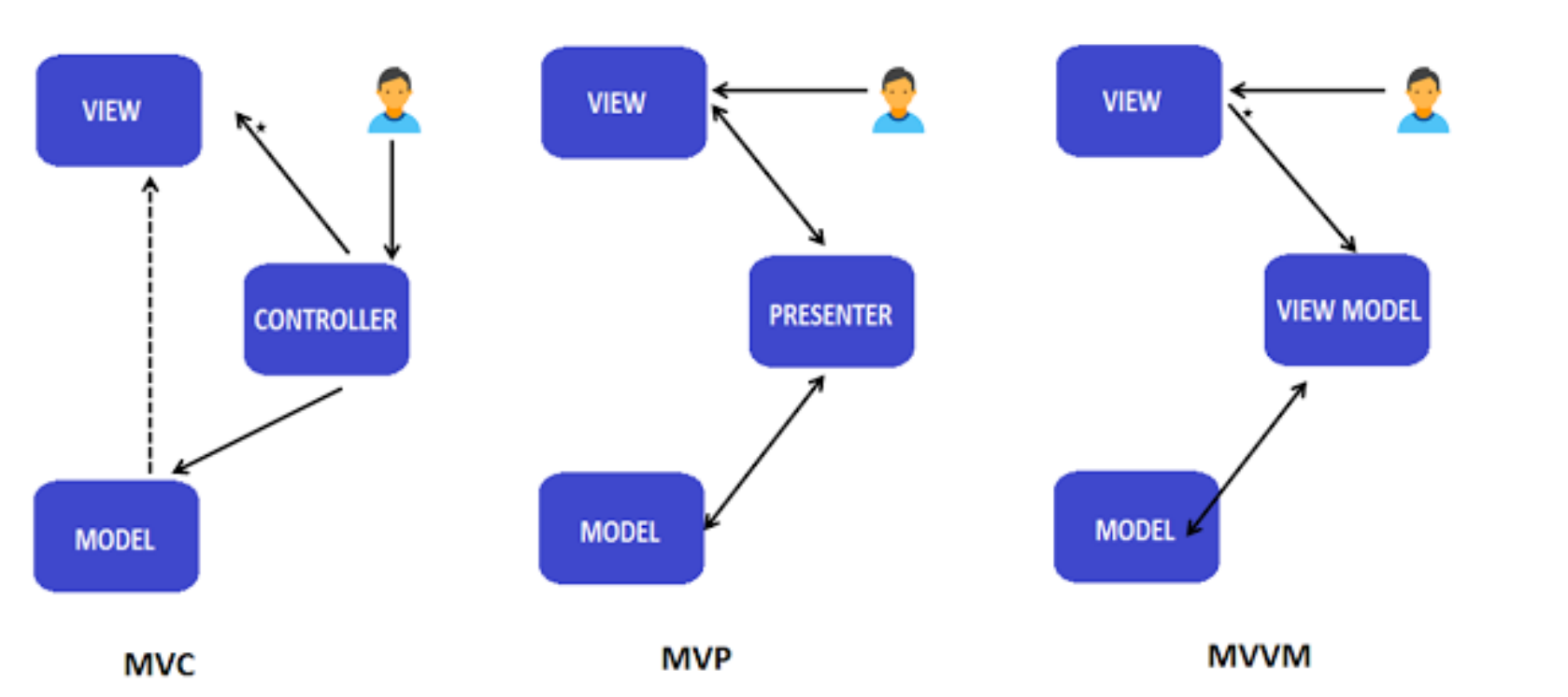
Доступ до даних

Шаблон MVVM

- Розкладемо код «по поличках»
  - Код простіше підтримувати
  - Краща тестованість
  - Краща розширюва-ність коду (extensibility)

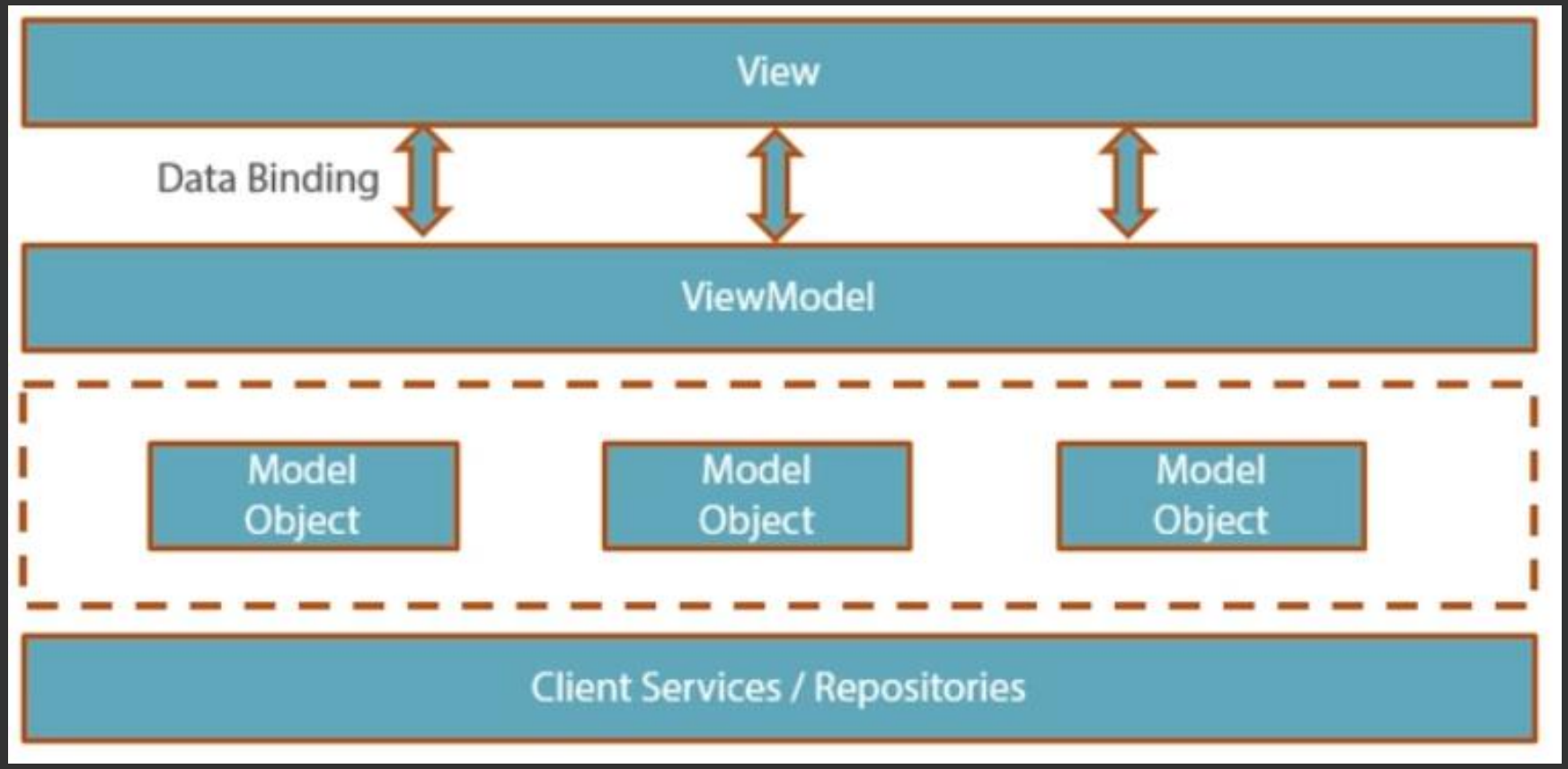


MVVM як результат еволюції



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

Шаблон MVVM у WPF



Рівень моделі

- Клієнтська 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.
- Дуже часто потрібно виконувати прив'язку до властивостей моделі, тому потрібно описувати сповіщення за допомогою події INotifyPropertyChanged.PropertyChanged.
- Інколи на рівень моделі виноситься валідація даних – у WPF це інтерфейси на зразок INotifyDataErrorInfo / IDataErrorInfo.

Рівень представлення (View)

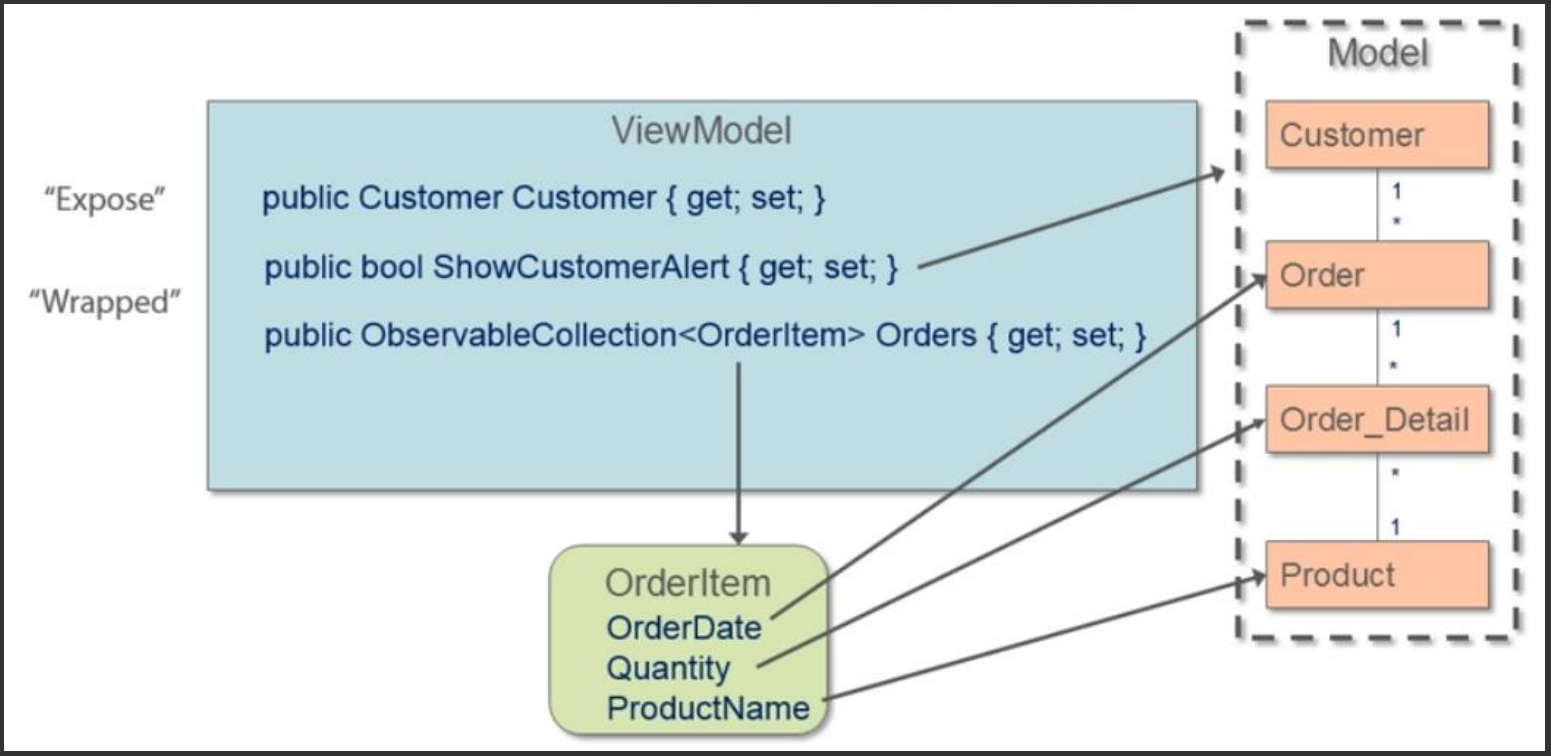
- Займається структуруванням того, що користувач бачить на екрані. Структура може складатись зі
  - Статичних частин – XAML-ієрархія, яка визначає елементи управління та їх макетування
  - Динамічних частин – анімацій, змін стану.
- Мета: описати представлення «без code behind» (C#).
  - Насправді такий код буде: ініціалізація, анімації, власна реалізація прив'язки даних тощо.
  - Ціль – зробити код більш декларативним, перенаправивши код прив'язування даних, реалізації команд, поведінок (behaviours) на рівень ViewModel.

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

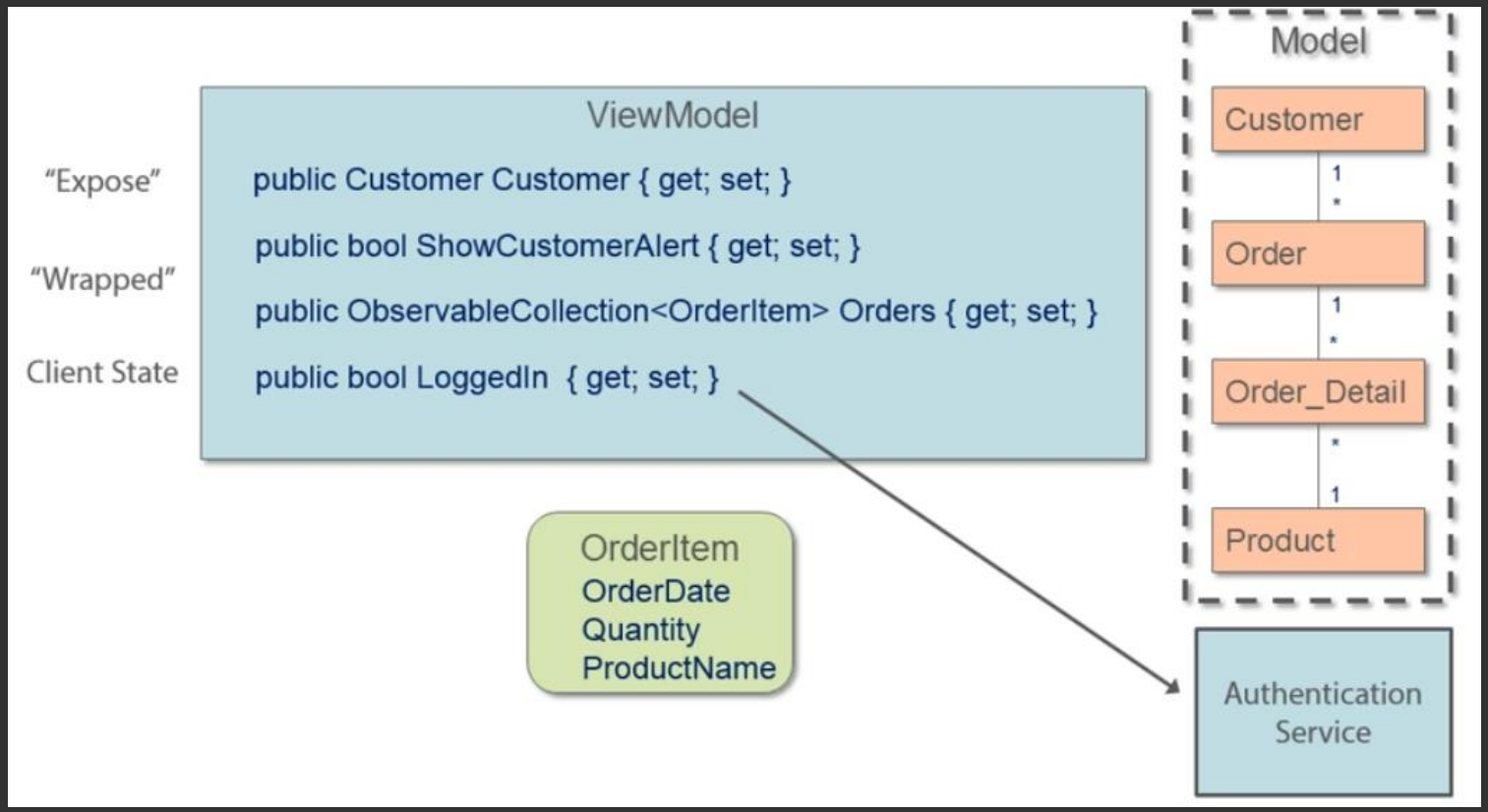
Дані на рівні ViewModel

- 1) Розкриття (expose) об'єктів з рівня моделі
- 2) Підготовка даних для представлення (wrap)



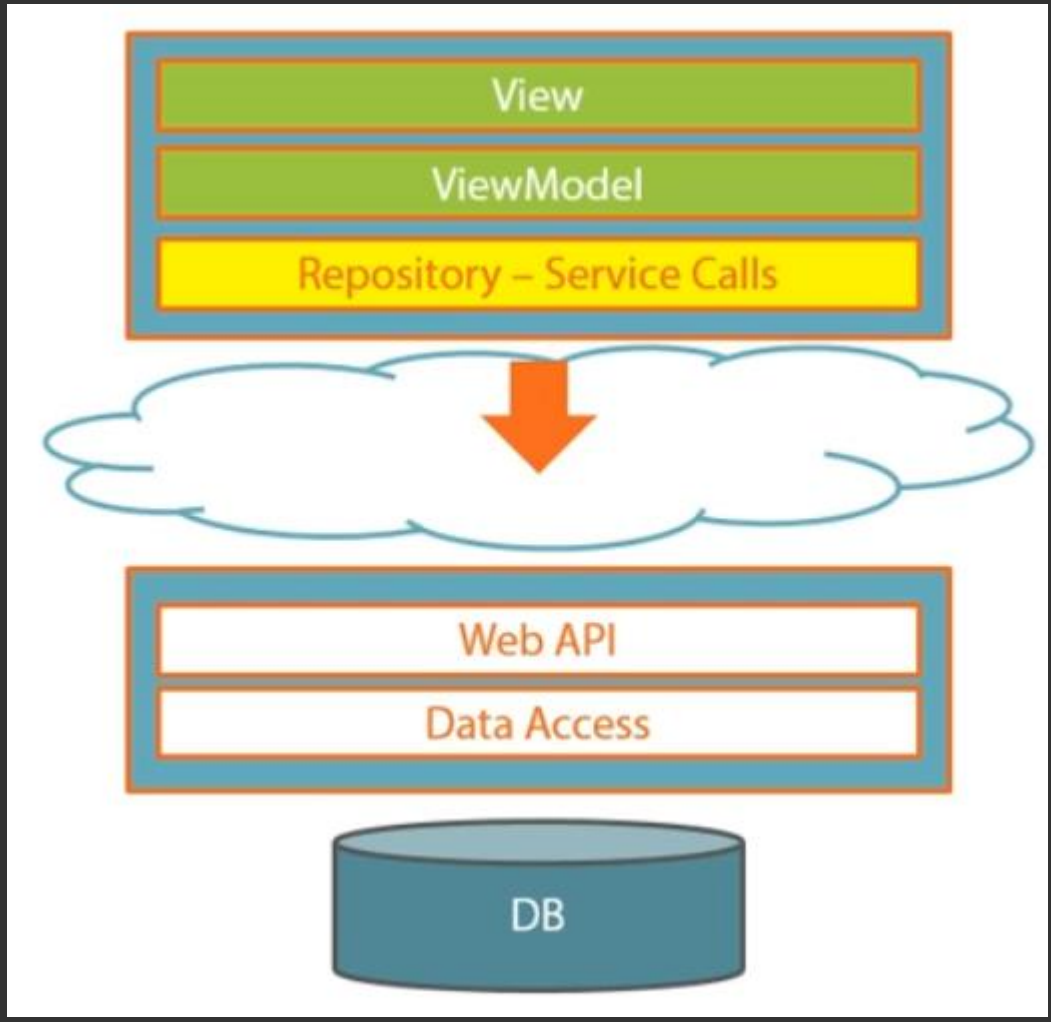
Дані на рівні ViewModel

- 3) Керування станом клієнтського додатку



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.



Фундаментальне рівняння MVVM

View.DataContext = ViewModel

View-First

- View is constructed first
- ViewModel gets constructed and attached to DataContext via View

ViewModel-First

- ViewModel is constructed first
- View is constructed as a consequence of ViewModel being added to UI

Підключення представлення до View Models

- 2) Використати вбудовану модель шаблонізації даних (data templating).
  - Інформує WPF щодо того, як рендерити об'єкти даних конкретного типу.

```
<DataTemplate DataType="{x:Type DataModels:User}">
  <TextBlock Text="{Binding Name}" />
</DataTemplate>
```

  - Властивість DataType визначає, до якого типу об'єкта (яких властивостей) потрібно мати доступ.
  - Для простоти виводиться Ім'я користувача (User.Name).
  - При прив'язці кількох об'єктів User до UI-елементу в області видимості DataTemplate кожен з них буде відображено фреймворком WPF (тут – TextBlock).

Підключення представлення до View Models

- Для кастомних об'єктів даних WPF шукає DataTemplate, оголошений для них.
  - Якщо знаходить, рендерить відповідно до внутрішньої XAML-розмітки.

```
<DataTemplate DataType="{x:Type ViewModels:UsersViewModel}">
  <Views:UsersView />
</DataTemplate>
```

  - Відбувається неявна прив'язка через властивість DataContext.
  - Незначний недолік підходу – потреба в додаванні нового DataTemplate в App.xaml для кожної пари View-ViewModel.
  - Метод прив'язки добре працює з підходом View Model first.

Підключення представлення до View Models (View Model first)

- Зазвичай використовується клас ContentControl із властивістю Content, прив'язаною до властивості ViewModel, which the application View Models are set to.

```
private BaseViewModel viewModel;

public BaseViewModel ViewModel
{
  get { return viewModel; }
  set { viewModel = value; NotifyPropertyChanged(); }
}

...

ViewModel = new UserViewModel();

...

<ContentControl Content="{Binding ViewModel}" />
```

Підключення представлення до View Models (View first)

- 3) Додаємо рівень абстракції – інтерфейси для кожної ViewModel-і.
  - Підхід дуже подібний до впровадження залежностей (dependency injection).
  - Фактично, можна використовувати DependencyManager для отримання подібного результату.

```
DependencyManager.Instance.Register<IUserViewModel, UserViewModel>();

...

public partial class UserView : UserControl
{
  public UserView()
  {
    InitializeComponent();
    DataContext = DependencyManager.Instance.Resolve<IUserViewModel>();
  }
}

...

<Views:UsersView />
```

Підключення представлення до View Models (View first)

- Недоліком підходу є створення інтерфейсу для всіх View Models разом з реєстрацією та обробкою (resolve) за допомогою DependencyManager.
- Основна відмінність від реалізації через View Model Locator полягає в тому, що локатор постачає рівень абстракції на основі шаблону «Одиночка».
  - Це дозволяє опосередовано інстанціювати View Models у XAML-коді, не використовуючи C#.

```
using CompanyName.ApplicationName.Managers;
using CompanyName.ApplicationName.ViewModels;
using CompanyName.ApplicationName.ViewModels.Interfaces;

namespace CompanyName.ApplicationName.Views.ViewModelLocators
{
  public class ViewModelLocator
  {
    public IUserViewModel UserViewModel
    {
      get { return DependencyManager.Instance.Resolve<IUserViewModel>(); }
    }
  }
}
```

```
<UserControl x:Class="CompanyName.ApplicationName.Views.UserView"
  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:ViewModelLocators="clr-namespace:
    CompanyName.ApplicationName.Views.ViewModelLocators"
  mc:Ignorable="d" Height="30" Width="300">
  <UserControl.Resources>
    <ViewModelLocators:ViewModelLocator x:Key="ViewModelLocator" />
  </UserControl.Resources>
  <UserControl.DataContext>
    <Binding Path="UserViewModel"
      Source="{StaticResource ViewModelLocator}" />
  </UserControl.DataContext>
  <TextBlock Text="{Binding User.Name}" />
</UserControl>
```

Коментарі до прикладу

- Екземпляр класу ViewModelLocator оголошено в секції ресурсів для View.
  - Спрацює лише при наявності безаргументного конструктора.
- View задає власну властивість DataContext у XAML, використовуючи BindingPath до властивості UserViewModel з ресурсу ViewModelLocator.
  - Потім властивість використовує DependencyManager для вибору (resolve) конкретної реалізації інтерфейсу IUserViewModel та повертає DependencyManager.

Інші переваги третього підходу

- Поширена проблема: конструктор Visual Studio не дозволяє в процесі дизайну отримати доступ до джерел даних та використовуваних інтерфейсів.
  - Для ViewModelLocator можна створити мок-ViewModels з фіктивними даними від властивостей, що допоможе візуалізувати представлення при конструюванні.
  - Можна використати прикріплену властивість IsInDesignMode з .NET-класу DesignerProperties.

```
public bool IsDesignTime
{
  get { return
    DesignerProperties.GetIsInDesignMode(new DependencyObject()); }
}
```

  - Об'єкт DependencyObject буде перевірятись.
  - Оскільки тут всі об'єкти повертають одне значення, кожного разу можна створювати новий об'єкт.

Інші переваги третього підходу

- Для спрощення коду можна зробити ViewModelLocator породженим від DependencyObject.
  - Успадкуються й інші небажані властивості.

```
using System.ComponentModel;
using System.Windows;
using CompanyName.ApplicationName.Managers;
using CompanyName.ApplicationName.ViewModels;
using CompanyName.ApplicationName.ViewModels.Interfaces;

namespace CompanyName.ApplicationName.Views.ViewModelLocators
{
  public class ViewModelLocator : DependencyObject
  {
    public bool IsDesignTime
    {
      get { return DesignerProperties.GetIsInDesignMode(this); }
    }

    public IUserViewModel UserViewModel
    {
      get
      {
        return IsDesignTime ? new MockUserViewModel() :
          DependencyManager.Instance.Resolve<IUserViewModel>();
      }
    }
  }
}
```

Узагальнений базовий клас для View Model Locator

```
using System.ComponentModel;
using System.Windows;
using CompanyName.ApplicationName.Managers;

namespace CompanyName.ApplicationName.Views.ViewModelLocators
{
  public abstract class BaseViewModelLocator<T> : DependencyObject
    where T : class
  {
    private T runtimeViewModel, designTimeViewModel;

    protected bool IsDesignTime
    {
      get { return DesignerProperties.GetIsInDesignMode(this); }
    }

    public T ViewModel
    {
      get { return IsDesignTime ?
        DesignTimeViewModel : RuntimeViewModel; }
    }
    protected T RuntimeViewModel
    {
      get { return runtimeViewModel ??
        (runtimeViewModel = DependencyManager.Instance.Resolve<T>()); }
    }

    protected T DesignTimeViewModel
    {
      set { designTimeViewModel = value; }
      get { return designTimeViewModel; }
    }
  }
}
```

- Одна властивість потрібна при роботі View Model, а інша – при дизайні.
- До третьої властивості View Model буде відбуватись прив'язка від Views.
  - Вона використовує властивість IsDesignTime, що визначити, яку View Model повертати.

Для кожної View model розробник має оголосити наступний код:

```
using CompanyName.ApplicationName.ViewModels;
using CompanyName.ApplicationName.ViewModels.Interfaces;

namespace CompanyName.ApplicationName.Views.ViewModelLocators
{
  public class UserViewModelLocator : BaseViewModelLocator<IUserViewModel>
  {
    public UserViewModelLocator()
    {
      DesignTimeViewModel = new MockUserViewModel();
    }
  }
}
```

```
<UserControl x:Class="CompanyName.ApplicationName.Views.UserView"
  ...
  <UserControl.Resources>
    <Locators:UserViewModelLocator x:Key="ViewModelLocator" />
  </UserControl.Resources>
  <UserControl.DataContext>
    <Binding Path="ViewModel" Source="{StaticResource ViewModelLocator}" />
  </UserControl.DataContext>
  ...
</UserControl>
```

Дякую за увагу!

Властивості класу Binding, що не використовуються в MVVM-додатках

For example, the three `NotifyOnSourceUpdated`, `NotifyOnTargetUpdated`, and `NotifyOnValidationError` properties relate to the raising of the `Binding.SourceUpdated`, `Binding.TargetUpdated` and `Validation.Error` Attached Events.

Likewise, the three `ValidatesOnDataErrors`, `ValidatesOnExceptions`, `ValidatesOnNotifyDataErrors` and `ValidationRules` properties all relate to the use of the `ValidationRule` class. This is a very UI-related way of validating, but this puts our business logic right into our Views component.

When using MVVM, we want to avoid this blending of components. We therefore tend to work with data elements rather than UI elements and so, we perform these kind of duties in our Data Model and/or View Model classes instead. We'll see this in `Chapter 8`, *Implementing Responsive Data Validation* later in the book, but now let's take a deeper look at the most important properties of the `Binding` class.