Промышленное программирование

Лекция 12. AvaloniaUI (часть 3: MVVM)

MVVM

Window.xaml (представление)

```
<Window>
    <StackPanel>
        <Label Content="{Binding Name}" />
        <CheckBox IsChecked="{Binding CanPlaceOrder}" />
        <ListBox Items="{Binding Orders}" />
    </StackPanel>
</Window>
```

ViewModel.cs (модель представления)

```
public sealed class ViewModel
{
    public string Name { get; } = string.Empty;
    public bool CanPlaceOrder { get; }
    public List<Order> Orders { get; } = new();
}
```

```
var view = new Window
{
    DataContext = new ViewModel()
};
```

Интерактивность ViewModel

Зачем нужна?

1. Нужна для View, если значения свойств могут изменяться в модели представления после отработки конструктора.
2. Нужна для ViewModel, если есть вычисляемые свойства.

Механизм реализации?

ViewModel должна реализовывать интерфейс INotifyPropertyChanged

```
namespace System.ComponentModel;

public class PropertyChangedEventArgs : EventArgs
{
    public virtual string? PropertyName { get; }

    public PropertyChangedEventArgs(string? propertyName);
}

public delegate void PropertyChangedEventHandler(object? sender, PropertyChangedEventArgs e);

public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler? PropertyChanged;
}
```

Интерактивность ViewModel

Мутабельные коллекции
Если в свойстве хранится коллекция, то как правило:
1) само свойство не изменяется (объект коллекции один и тот же)
2) меняется содержимое объекта коллекции.

Механизм реализации:
Тип коллекции должен реализовывать интерфейс INotifyCollectionChanged

```
namespace System.Collections.Specialized;

public class NotifyCollectionChangedEventArgs : EventArgs
{
    public NotifyCollectionChangedAction Action { get; }
    public IList? NewItems { get; }      // The items affected by the change
    public IList? OldItems { get; }      // The old items affected by the change (for Replace events)
    public int NewStartingIndex { get; } // The index where the change occurred
    public int OldStartingIndex { get; } // The old index where the change occurred (for Move events)
}

public delegate void NotifyCollectionChangedEventHandler(object? sender, NotifyCollectionChangedEventArgs e);

public interface INotifyCollectionChanged
{
    event NotifyCollectionChangedEventHandler? CollectionChanged;
}
```

Вычисляемые свойства

```
public sealed class UserViewModel : INotifyPropertyChanged
{
    public string Surname { … }             // Иванов
    public string Name { … }                // Сергей
    public string Patronymic { … }          // Петрович

    public string SurnameAndInitials { … } // Иванов С. П.

    private static string GetSurnameAndInitials(string surname, string name, string patronymic)
    {
        …
    }

    …
}
```

Вычисляемые свойства: решение “в лоб”

```
public sealed class UserViewModel : INotifyPropertyChanged
{
    …

    public UserListViewModel()
    {
        PropertyChanged += (sender, args) =>
        {
            if (args.PropertyName == nameof(Surname) ||
                args.PropertyName == nameof(Name) ||
                args.PropertyName == nameof(Patronymic))
            {
                SurnameAndInitials = GetSurnameAndInitials(Surname, Name, Patronymic);
            }
        };
    }
}
```

IObservable<T>, IObserver<T>

```
namespace System;

public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

public interface IObserver<in T>
{
    void OnNext(T value);
    void OnCompleted();
    void OnError(Exception error);
}
```

Extensions

```
namespace System;

public static class ObservableExtensions {
    public static IDisposable Subscribe<T>(this IObservable<T> source, Action<T> onNext);
    …
}
```

```
namespace System.Reactive.Linq;

public static class Observable {
    public static IObservable<TResult> CombineLatest<T1, T2, TResult>(this IObservable<T1> first,
                                                                       IObservable<T2> second,
                                                                       Func<T1, T2, TResult> resultSelector);
    public static AsyncSubject<T> GetAwaiter<T>(this IObservable<T> source)
    …
}
```

ReactiveUI, ReactiveUI.Fody

Руководство: https://www.reactiveui.net/docs/handbook/

NuGet-пакеты, зависимые от фреймворка:
Avalonia.ReactiveUI

NuGet-пакеты, независимые от фреймворка:
ReactiveUI.Fody

Для включения поддержки Rx в Avalonia:

```
public static AppBuilder BuildAvaloniaApp()
{
    return AppBuilder
        .Configure<App>()
        .UseReactiveUI()
        …;
}
```

ObservableAsPropertyHelper (OAPH)

```
public sealed class ViewModel : ReactiveObject
{
    [Reactive] public string Surname { get; set; } = string.Empty;
    [Reactive] public string Name { get; set; } = string.Empty;
    [Reactive] public string Patronymic { get; set; } = string.Empty;

    private readonly ObservableAsPropertyHelper<string> _surnameAndInitials;
    public string SurnameAndInitials => _surnameAndInitials.Value;

    public void ViewModel()
    {
        // IObservable<string> surname = this.WhenAnyValue(viewModel => viewModel.Surname);

        // IObservable<(string, string)> surnameAndName =
        //     this.WhenAnyValue(viewModel => viewModel.Surname, viewModel => viewModel.Name);

        IObservable<string> surnameAndInitials = this.WhenAnyValue(
            viewModel => viewModel.Surname,
            viewModel => viewModel.Name,
            viewModel => viewModel.Patronymic,
            GetSurnameAndInitials);

        _surnameAndInitials = surnameAndInitials.ToProperty(this, viewModel => viewModel.SurnameAndInitials);
    }
}
```

ReactiveCommand

```
<Window>
    <Button Command="{Binding LoginCommand}" />
</Window>
```

```
public sealed class ViewModel : ReactiveObject
{
    public ReactiveCommand<Unit, Unit> LoginCommand { get; }

    public ViewModel()
    {
        IObservable<bool> canLogin = …;
        LoginCommand = ReactiveCommand.Create(Login, canLogin);
        // (!) CreateFromTask
    }

    private void Login() { … }
}
```

Interaction (Code-Behind)

```
public sealed class ViewModel : ReactiveObject
{
    public ReactiveCommand<Unit, Unit> LoginCommand { get; }

    public Interaction<string, Unit> ShowError { get; } = new();

    public ViewModel()
    {
        LoginCommand = ReactiveCommand.CreateFromTask(LoginImpl);
    }

    private async Task LoginImpl()
    {
        …
        await ShowError.Handle("Something went wrong :(");
        …
    }
}
```

Interaction (XAML)

```
// MainWindow.axaml.cs
public partial class MainWindow : ReactiveWindow<ViewModel>
{
    public MainWindow()
    {
        InitializeComponent();

        this.WhenActivated(cd =>
        {
            if (ViewModel is null)
                return;

            cd.Add(ViewModel.ShowError.RegisterHandler(async ctx =>
            {
                // ctx.Input, ctx.SetOutput(…)
            }));
        });
    }
}
```

Обзор

Представление (View)

Window.xaml
– Свойства-значения привязываются к реактивным свойствам в ViewModel.
– Свойства-коллекции (Items у DataGrid) привязываются к ObservableCollection<T> в ViewModel.
– Свойства-команды (Command у Button) привязываются к ReactiveCommand<TInput, TOutput> в ViewModel.

Window.xaml.cs
– Объявляются методы-обработчики для Interaction-ов у ViewModel.
– В конструкторе эти методы регистрируются как обработчики Interaction-ов у конкретного объекта ViewModel.

Используется пакет Avalonia.

Модель представления (ViewModel)

– Свойства-значения, которые могут меняться, помечаются атрибутом [Reactive].
– Для свойства-коллекции, содержимое которой может меняться, используется тип ObservableCollection<T>.
– Метод, который должен вызываться при нажатии на кнопку, оборачивается в ReactiveCommand<TInput, Toutput>.

– Для вычисляемых свойств используется IObservable<T>.
– Для управления доступностью (“включенности”) ReactiveCommand<TInput, TOutput> используется IObservable<bool>.
– Свойство можно превратить в IObservable<T> через this.WhenAnyValue.
– IObservable<T> можно превратить в свойство через OAPH (метод расширения ToProperty).

Используется пакет ReactiveUI. ⚠ **Не используется ничего, зависящее от конкретного UI-фреймворка (ничего из пакета Avalonia).**
При необходимости взаимодействия с UI-фреймворком (показать новое окно, закрыть окно и т. п.) используется Interaction<TInput, TOutput>.
При необходимости выполнить какой-то код в UI-потоке используется RxApp.MainThreadScheduler.