Зачем?

Программа предназначена для упрощения кода оконных приложений windows на платформе WPF с использованием MVVM Light.

Без использования ApplicationPatcher.Wpf код выглядит так:

1. Свойства ViewModel:

private int anyValue;

public int AnyValue {

get => anyValue;

set => Set(() => AnyValue, ref anyValue, value);

}

1. Команды ViewModel:

private ICommand anyActionCommand;

public ICommand AnyActionCommand => (anyActionCommand ?? new RelayCommand(AnyActionMethod, CanAnyActionMethod))

public void AnyActionMethod() {

...

}

public bool CanAnyActionMethod() {

...

}

1. Свойства UserControl:

public static readonly DependencyProperty MyValueProperty = DependencyProperty.Register("MyValue", typeof(string), typeof(UserControl1));

public string MyValue {

get { return (string)GetValue(MyValueProperty); }

set { SetValue(MyValueProperty, value); }

}

C использованием ApplicationPatcher.Wpf:

1. Свойства ViewModel:

public int AnyValue { get; set; }

1. Команды ViewModel:

public void AnyActionMethod() {

...

}

public bool CanAnyActionMethod() {

...

}

1. Свойства UserControl:

public string MyValue { get; set; }

Как?

Вся магия происходит после компиляции WPF приложения, для этого необходимо в post-build event написать вызов приложения ApplicationPatcher.Wpf с указанием пути для приложения:

call ...\ApplicationPatcher.Wpf\ApplicationPatcher.Wpf.exe $(TargetPath)

Тем самым ApplicationPatcher.Wpf зайдет в скомпилированную .net сборку и сделает всю рутинную работу.

Для более точной настройки можно использовать атрибуты из ApplicationPatcher.Wpf.Types (об этом дальше).

Без атрибутов будут использоваться параметры из конфигурации ApplicationPatcher.Wpf.config.xml такие как правила именования членов, типы выбора классов и членов класса для патчинга.

Далее будут сформированы группы членов класса: свойства будут искать для себя поля, методы команд будут искать для себя can execute методы и свойства с их полями. После недостающие члены будут созданы и связаны с помощью переписывания il кода.

Обо всем, что сделал патчер, можно прочитать в файле логов ApplicationPatcher.Wpf.log. Там будут написано какие классы патчер взял для патчинга и какие группы в них нашел.

Настройка

Изначально члены класса будут пытаться объединится в группы с помощью правила формирования имен, указанных в конфигурации ApplicationPatcher.Wpf.config.xml.

Правило имени состоит из префикса, суффикса и типа имени. Типов имен всего 5: all\_lower, ALL\_UPPER, First\_upper, lowerCamelCase, UpperCamelCase. Из названия должно быть понятно как себя ведет каждый из типов.

Таким образом, чтобы создавать приватные поля типа \_myPrivateField, нужно указать тип имени lowerCamelCase с префиксом “\_”.

Так же можно явно указать с чем связывать член класса.

Свойства

1. PatchingPropertyAttribute – патчить это поле.
2. NotPatchingPropertyAttribute – не патчить это поле.
3. NotUseSearchByNameAttribute – не использовать поиск по имени.
4. ConnectPropertyToFieldAttribute(string connectingFieldName) – связать свойство с указанным полем.
5. ConnectFieldToPropertyAttribute(string connectingPropertyName) – связать поле с указанным свойством.

Команды

1. PatchingCommandAttribute – патчить эту команду.
2. NotPatchingCommandAttribute – не патчить эту команду.
3. NotUseSearchByNameAttribute - не использовать поиск по имени.
4. ConnectMethodToMethodAttribute(string connectingMethodName) – связать метод с указанным методом.
5. ConnectMethodToPropertyAttribute(string connectingPropertyName) – связать метод с указанным свойством.
6. ConnectPropertyToMethodAttribute(string connectingMethodName) или ConnectPropertyToMethodAttribute(string connectingFirstMethodName, string connectingSecondMethodName) – связать поле с одним или двумя методами.

Таким образом, чтобы создать группу команды из execute method AnyActionMethod, can execute method CanAnyMySecondActionMethod, property ThisIsProperty и field thisIsProperty, нужно написать такой код:

[ConnectMethodToMethod(nameof(CanAnyMySecondActionMethod))]

[ConnectMethodToProperty(nameof(ThisIsProperty))]

public void AnyActionMethod() {

...

}

public bool CanAnyMySecondActionMethod() {

...

}

public ICommand ThisIsProperty { get; set; }

private ICommand thisIsProperty;

Либо можно указать connect от property к методам, это тоже будет работать, либо от can execute method к execute method и связать execute method с property любым способом.

Так же можно не писать field thisIsProperty, оно будет создано автоматически. Можно не писать ни property ThisIsProperty ни field thisIsProperty, они оба могут быть созданы автоматически, но тогда имена будут сконвертированы из имени execute method и будут AnyActionCommand и anyActionCommand.

Для каждой ViewModel можно указать атрибуты:

1. PatchingViewModelAttribute(ViewModelPatchingType patchingType) – патчить эту ViewModel с выбранным типом патчинга.
2. NotPatchingViewModelAttribute – не патчить эту ViewModel.

Для каждого UserControl можно указать атрибуты:

1. PatchingFrameworkElementAttribute(FrameworkElementPatchingType patchingType) – патчить этот UserControl с выбранным типом патчинга.
2. NotPatchingFrameworkElementAttribute – не патчить этот UserControl.

Так же в атрибутах проекта можно указать SelectingViewModelAttribute(ViewModelSelectingType selectingType) и SelectingFrameworkElementAttribute(FrameworkElementSelectingType selectingType), где задать правило выбора классов ViewModel и UserControl.

Приоритетно берется значение правила выбора из атрибутов, потом из конфигурации.

Дополнительно в конфигурации есть:

1. SkipConnectingByNameIfNameIsInvalid – если имя члена некорректное, то можно либо пропустить поиск по имени, либо выдать ошибку.
2. ConnectByNameIfExsistConnectAttribute – использовать ли поиск по имени если у члена присутствует какой-либо connect атрибут.