Implementing WPF dependency properties with Metalama

blog.postsharp.net/wpf-dependency-property-metalama

Darío Macchi

November 21, 2024

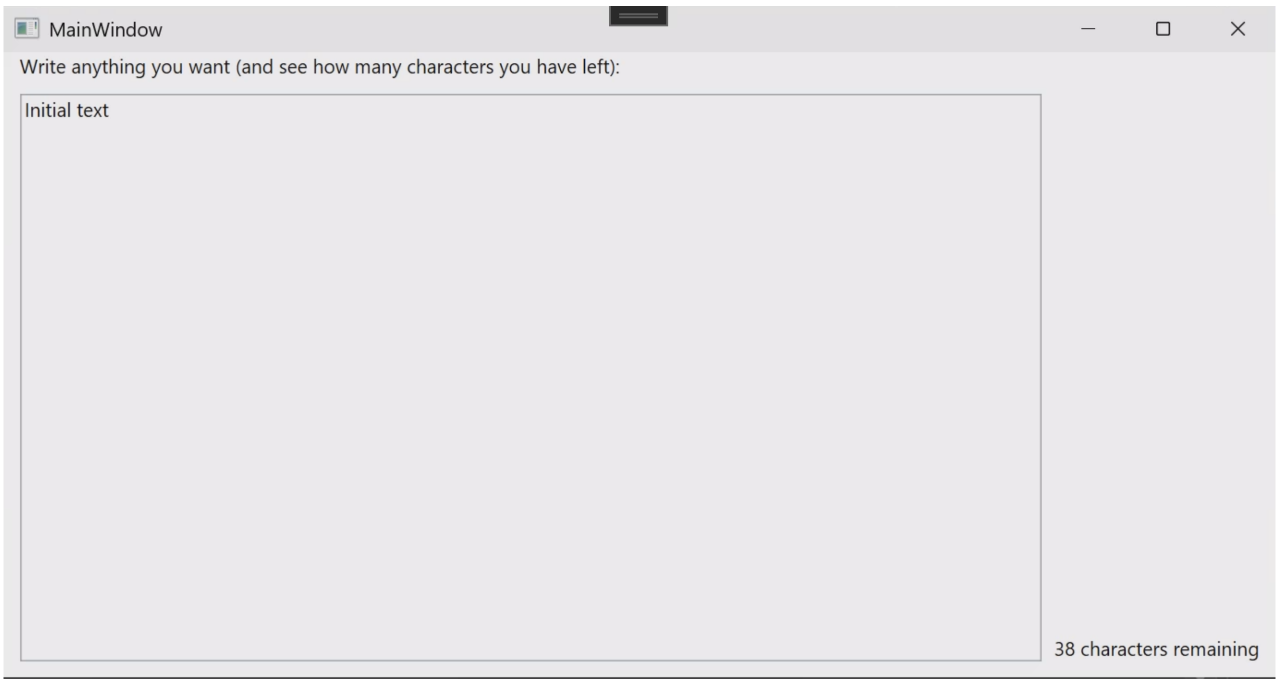
When building user controls in WPF, it's best practice to expose dependency properties in addition to normal C# properties. Unfortunately, implementing custom dependency properties requires a fair amount of redundant code. In this article, we'll explore how to use Metalama to eliminate this boilerplate code.

Dependency properties allow WPF to assign these properties to a *source* of values, enabling the UI to refresh when the source changes or implement animations – a mechanism called *data binding*. In contrast, C# properties are directly assigned to a *value*, one time. The downside of dependency properties is that implementing them manually can be tedious and error-prone. It requires writing significant boilerplate code to register the property (using `DependencyProperty.Register`) and manage property-changed and validation callbacks.

In this article, we'll show how to reduce redundant code using Metalama, a powerful tool that automates repetitive coding tasks using aspects, thus simplifying the creation of custom dependency properties. This reduces development time and improves code consistency. Specifically, we'll demonstrate the `[DependencyProperty]` aspect and show how to add validation and callbacks.

Example app

In this article, we'll use the example of a simple custom control called `LimitedTextBox`. This control has two dependency properties: `MaxLength` and `Text`. The `MaxLength` property specifies the maximum number of characters allowed in the text box, while the `Text` property holds the text entered by the user. As the user types into the `LimitedTextBox` control, it automatically updates the counter showing the number of characters remaining to reach the limit.



Write anything you want (and see how many characters you have left):

Initial text

38 characters remaining

If you were to implement the `MaxLength` dependency manually, you'd end up with the following *three* snippets:

```
public static readonly DependencyProperty MaxLengthProperty =
DependencyProperty.Register(
    nameof(MaxLength),
    typeof(int),
    typeof(LimitedTextBox),
    new PropertyMetadata( 100, OnMaxLengthChanged ),
    ValidateMaxLength );

public int MaxLength
{
    get => (int) this.GetValue( MaxLengthProperty );
    set => this.SetValue( MaxLengthProperty, value );
}

private static void OnMaxLengthChanged(
    DependencyObject d,
    DependencyPropertyChangedEventArgs e )
{
    var control = (LimitedTextBox) d;
    control.UpdateRemainingCharsText( control._textBox.Text );
}

private static bool ValidateMaxLength( object value ) => value is > 0;
```

```
{}
```

The full source code of examples in this article is available on GitHub.

Aside from the complexity of the `DependencyProperty.Register` method, you can see how this manual implementation can easily lead to errors and inconsistencies, especially as the number of properties grows. This is where Metalama comes in to simplify the process and reduce the amount of manual work required to implement dependency properties.

Let’s see how we can simplify this using Metalama.

Implementing dependency properties with Metalama

Metalama is a tool that facilitates real-time code generation and validation in C# through the use of aspects. Aspects are special classes that work within the compiler to dynamically transforms code when you build, never committing the changes to your source code. This tool helps automate the creation of repetitive code, such as implementing dependency properties, INotifyPropertyChanged, WPF commands, and many others.

If you need to generate boilerplate code for a specific situation (like this one), you can create an aspect from scratch for it. However, as this task is quite common among WPF developers, Metalama simplifies it by offering a built-in solution.

The `[DependencyProperty]` aspect is one of the many open-source, production-ready aspects provided by Metalama. This aspect is specifically designed to automate the generation of the boilerplate code needed to implement dependency properties while maintaining flexibility. If you’re interested in exploring more of these aspects, be sure to check out the Metalama Marketplace.

Basically, the `[DependencyProperty]` turns a plain old C# automatic property into a dependency property.

To use the `[DependencyProperty]` aspect in your project, you must:

1. Add the Metalama.Patterns.Wpf package to your project.
2. Add the `[DependencyProperty]` custom attribute to a standard C# automatic property. Note that the containing type of the property must be derived from `DependencyObject`.

Let’s see how the `MaxLength` property can be implemented using the `[DependencyProperty]` aspect:

```
[DependencyProperty]
public int MaxLength { get; set; } = 100;
```

In the code snippet above, we use the `[DependencyProperty]` aspect to decorate the `MaxLength` property in the `LimitedTextBox` class. Note that the property should be auto-implemented (no backing field required), and the default value is set directly in the property declaration. The aspect takes care of generating the necessary boilerplate code, including the property registration, metadata, and validation callbacks.

You can take a look at what the generated code will look like using our Metalama Diff tool (included in Visual Studio Tools for Metalama).

Metalama Diff tool

By using the `[DependencyProperty]` aspect, we eliminate the need to manually implement the dependency property registration, metadata, and validation callbacks. This approach significantly reduces the amount of boilerplate code and ensures consistency across different dependency properties in the project.

Adding validation with an attribute

If you've implemented dependency properties manually, you're likely familiar with *validation callback* methods. With Metalama, validating a property can often be done using a simple *contract* custom attribute from the Metalama.Patterns.Contracts package.

Some examples are the [Email], [Phone], and [Url], or [NotEmpty] contracts.

Here you can see an example where we apply the [StrictlyGreaterThan] contract to the `MaxLength` property:

```
[StrictlyGreaterThan( 0 )]

[DependencyProperty]
public int MaxLength { get; set; } = 100;
```

This approach results in compact and readable source code.

Adding a validation callback

Let's now turn to the second dependency property: `Text`. We want to validate that the text is only made of letters or whitespaces. Although we could implement this requirement by using the [RegularExpression] contract, we'll show here how to do this using a callback method.

Validation callbacks are methods that run *before* the property is set. If they fail, the property is not set. There are two ways to add a validation contract:

- *Implicitly* by following a *naming convention* and creating a method whose name corresponds to the property name, plus the `Validate` prefix. In this case, the property name is `Text`, so the validation method should be named `ValidateText`.

- *Explicitly*, by setting the `ValidateMethod` parameter of the `[DependencyProperty]` type.

Metalama supports several signatures for the validation callback.

Here is the validation callback for the `Text` property:

```
private void ValidateText( string value )
{
    if ( !string.IsNullOrWhiteSpace( value )
         && !value.All( c => char.IsLetter( c ) || char.IsWhiteSpace( c ) ) )
    {
        throw new ArgumentException(
            "Invalid Text value. Only letters and whitespace are allowed." );
    }
}

[DependencyProperty(ValidateMethod = "WhateverValidationMethodNameYouWant")]
public string Text { get; set; }
```

Unlike the method used in the `DependencyProperty.Register`, the validation method used by Metalama doesn’t return a boolean value; instead, it throws an exception if the value is invalid.

Adding a PropertyChanged callback

Property-changed callbacks are invoked *after* the value of a dependency property has changed. As with the validation callback, there are two ways to specify it:

- *Implicitly* by following a naming convention. For example, the name of our property is `MaxLength`, so the PropertyChanged method should be named `OnMaxLengthChanged`. The same applies to the `Text` property and the `OnTextChanged` method. Metalama will automatically detect and use them as property-changed callbacks.
- *Explicitly* by setting the `PropertyChangedMethod` of the `[DependencyProperty]` attribute.

As with the validation callback, there are several signatures for the `PropertyChanged` method (see them in the documentation here), so you can choose the one that best fits your needs. An important detail compared to the method used in the manual implementation (using the `DependencyProperty.Register`) is that the `PropertyChanged` method does not need to be static and can access the instance of the class.

Here are the `PropertyChanged` methods for our dependency properties:

```
private void OnMaxLengthChanged( int oldValue, int newValue )
{
    this.UpdateRemainingCharsText( this._textBox.Text );
}

private void OnTextChanged( string oldValue, string newValue )
{
    this.UpdateRemainingCharsText( newValue );
}
```

Why use the Metalama approach?

The Metalama approach to implementing dependency properties offers several advantages over the manual approach. Here are some key benefits.

Improved code readability and maintainability

By using the `[DependencyProperty]` aspect, you can eliminate the (ugly) boilerplate code typically associated with dependency property registration. This results in cleaner, more concise code that is easier to read and maintain.

The use of idiomatic C# code with aspects makes it easier for developers to understand the purpose of the code and its intended behavior.

Enhanced developer productivity

By leveraging Metalama, developers can focus on more critical aspects of their application, rather than getting bogged down in repetitious tasks. The automation provided by Metalama allows for quicker implementation of common patterns, leading to faster development cycles and more reliable code. The Metalama approach significantly reduces the amount of manual work needed to implement dependency properties. This isn’t just due to the reduction in boilerplate code but also because contracts from the `Metalama.Patterns.Contracts` package can help avoid reinventing the wheel by providing common validation methods.

This approach not only minimizes the likelihood of errors but also makes code maintenance and readability easier. By automatically generating the necessary code using aspects like `DependencyProperty` and contracts, developers can focus more on core functionality rather than repetitive code. As a result, development processes are sped up, and overall efficiency in software projects is improved.

Conclusion

Manually implementing custom dependency properties in WPF can be a complex and error-prone task. It demands meticulous attention to detail, which can be time-consuming and may lead to inconsistencies or errors if not managed carefully.

However, tools like Metalama can significantly streamline this process. By using the `[DependencyProperty]` aspect, developers can automate the generation of the required code to implement dependency properties. This automation ensures consistency and reduces the potential for errors, simplifying the development of custom controls. It also allows developers to concentrate more on the core functionality of their applications, rather than getting bogged down by repetitive coding tasks.

Related articles

- Implementing custom dependency properties in WPF (+example)
- Implement INotifyPropertyChanged with Metalama
- Implement ICommand with Metalama
- More from the Timeless .NET Engineer series