Implement ICommand with Metalama

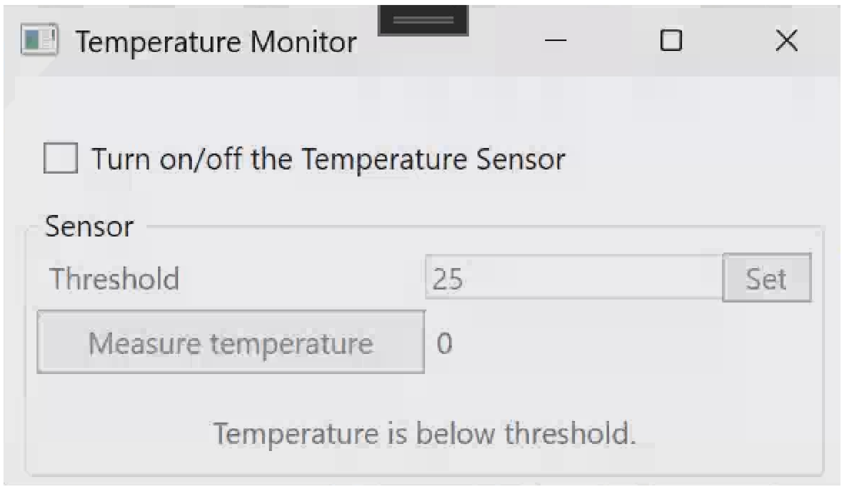
blog.postsharp.net/wpf-command-metalama

Darío Macchi November 4, 2024

This article explores the use of the `ICommand` interface in a temperature monitor application, contrasting manual and automated approaches using Metalama. Initially, it details how to manually create commands for operating a temperature sensor, highlighting the repetitive coding and potential for errors. Then, it introduces Metalama and its `[Command]` aspect to automatically generates the necessary boilerplate code for `ICommand` implementations. The comparison reveals that Metalama minimizes manual effort and errors, enhancing development efficiency.

ICommand manual implementation - an example

As seeing is believing, we will use a simple example to show how to implement the `ICommand` interface manually before implement it using Metalama. This example is about a temperature monitor application that allows the user to set a threshold and get the temperature from the sensor to check if it is above the threshold or not.


Temperature Monitor
Turn on/off the Temperature Sensor
Sensor
Threshold
25
Set
Measure temperature
0
Temperature is below threshold.

This application has three different commands of varying complexity:

1. A command to toggle the temperature sensor on or off.
2. A command to set the sensor's threshold temperature.
3. A command to obtain the current temperature from the sensor.

Despite each command's specific complexity (due to this particular example's business logic), they all share a common structure:

- Each command is encapsulated in a class implementing the `ICommand` interface.
- Every command includes a `CanExecute` method, which returns a boolean indicating whether execution is possible.
- Each command has an `Execute` method that performs its designated action.
- Commands can trigger a `CanExecuteChanged` event to inform the UI of any changes in their executability (`CanExecute` status).

Let's examine the simplest command: **toggling the sensor on or off**.

```
internal sealed class ToggleTemperatureSensorCommand : ICommand
{
    private readonly TemperatureSensor _sensor;

    public ToggleTemperatureSensorCommand( TemperatureSensor sensor )
    {
        this._sensor = sensor;
    }

    public event EventHandler? CanExecuteChanged;

    public bool CanExecute( object? parameter )
    {
        return true;
    }

    public void Execute( object? parameter )
    {
        this._sensor.IsEnabled = !this._sensor.IsEnabled;
    }
}
```

This command requires no parameters, and its `CanExecute` method always returns `true`, as toggling is always possible. Consequently, the `Execute` method's sole task is changing the sensor's state. The `CanExecuteChanged` event isn't used here since execution is perpetually feasible.

Next step? Simply define a public property like this in your view model:

```
public ICommand ToggleTemperatureSensorCommand { get; }
```

and then initialize it in the constructor like this:

```
this.ToggleTemperatureSensorCommand = new ToggleTemperatureSensorCommand(
this.Sensor );
```

The last part is to bind this command to a UI component in our view, such as a checkbox for toggling purposes. We bound it to the `ToggleTemperatureSensorCommand` using the `Command` attriute to easily turn the sensor on or off and enable or disable the GroupBox accordingly.

```
<CheckBox
  Content="Turn on/off the Temperature Sensor"
  Command="{Binding ToggleTemperatureSensorCommand}"
/>
```

Before jumping to the Metalama implementation, let's analyze a hand-written implementation of `SetThresholdCommand`:

```
internal sealed class SetThresholdCommand : ICommand
{
    private readonly TemperatureSensor _sensor;

    public SetThresholdCommand( TemperatureSensor sensor )
    {
        this._sensor = sensor;

        // Subscribe to sensor property changes
        this._sensor.PropertyChanged += this.OnSensorPropertyChanged;
    }

    public event EventHandler? CanExecuteChanged;

    public bool CanExecute( object? parameter )
    {
        return this._sensor.IsEnabled;
    }

    public void Execute( object? parameter )
    {
        this._sensor.Threshold = Convert.ToDouble( parameter!,
CultureInfo.CurrentCulture );
    }

    private void OnSensorPropertyChanged( object? sender, PropertyChangedEventArgs
e )
    {
        if ( e.PropertyName is null or nameof(TemperatureSensor.IsEnabled) )
        {
            // Notify WPF that CanExecute has changed
            this.CanExecuteChanged?.Invoke( this, EventArgs.Empty );
        }
    }
}
```

This command takes a parameter to set the threshold temperature.

- The `CanExecute` method returns `true` only if the sensor is turned on.
- The `Execute` method sets the threshold temperature using the value of the parameter sent by the UI.
- The `CanExecuteChanged` event is triggered when the `OnPropertyChanged` event is fired by the `IsEnabled` property of the sensor, which occurs when the sensor is turned off.

And how the parameter is passed from the UI to the command? To pass a parameter from the user interface to a command, we should use the `CommandParameter` property of the component in the view. In our example, this is applied to the `Set` button.

```
<Button Content="Set" Command="{Binding SetThresholdCommand}" CommandParameter="
{Binding Threshold}" />
```

In the two examples we analyzed (`ToggleTemperatureSensorCommand` and the `SetThresholdCommand`), implementing the commands manually required a lot of repetitive code, increasing the likelihood of errors as you move away from the UI. So, our goal using Metalama is to reduce the amount of manual code required to implement these commands and automating the repetitive tasks.

Implementing ICommand with Metalama

Metalama is a tool designed for on-the-fly code generation and validation in C# using aspects, special classes that run in the compiler or the IDE and dynamically transform the source code. It automates the creation of boilerplate code, such as the implementation of the `ICommand` interface. You have the option to implement an aspect from scratch to generate boilerplate code for this interface. However, because this is such a common pattern among .NET developers, Metalama simplifies the process by offering a built-in solution.

The `[Command]` aspect is one of the many open-source, production-ready aspects provided by Metalama. This aspect is specifically designed to automate the generation of boilerplate code for the `ICommand` interface while maintaining flexibility. For more information, you can explore additional aspects on the Metalama Marketplace.

Let see the most basic way to apply it to our Temperature Monitor example, starting with the command to toggle the temperature sensor:

1. First, you need to add the `Metalama.Patterns.WPF` package to your project.
2. Then, you have to define a `ToggleTemperatureSensor` method that will execute the toggle action.
3. Finally, the only thing you need to do is to apply the `[Command]` aspect to the `ToggleTemperatureSensor` method.

Here you have the `ToggleTemperatureSensorCommand` implemented with Metalama:

```
[Command]
public void ToggleTemperatureSensor()
{
    this.Sensor.IsEnabled = !this.Sensor.IsEnabled;
}
```

When the `[Command]` attribute is applied to a method, the aspect automatically generates a property following this pattern: method name plus the *Command* word; that’s why we are talking about the `ToggleTemperatureSensorCommand`.

What about the `CanExecute` method? The aspect may automatically generates a `CanExecute` method for you if you want. You just need to define a boolean property or method that returns a boolean value with a name that follow one of these rules:

- `Can` + method name (e.g. `CanToggleTemperatureSensor`)
- `CanExecute` + method name (e.g. `CanExecuteToggleTemperatureSensor`)

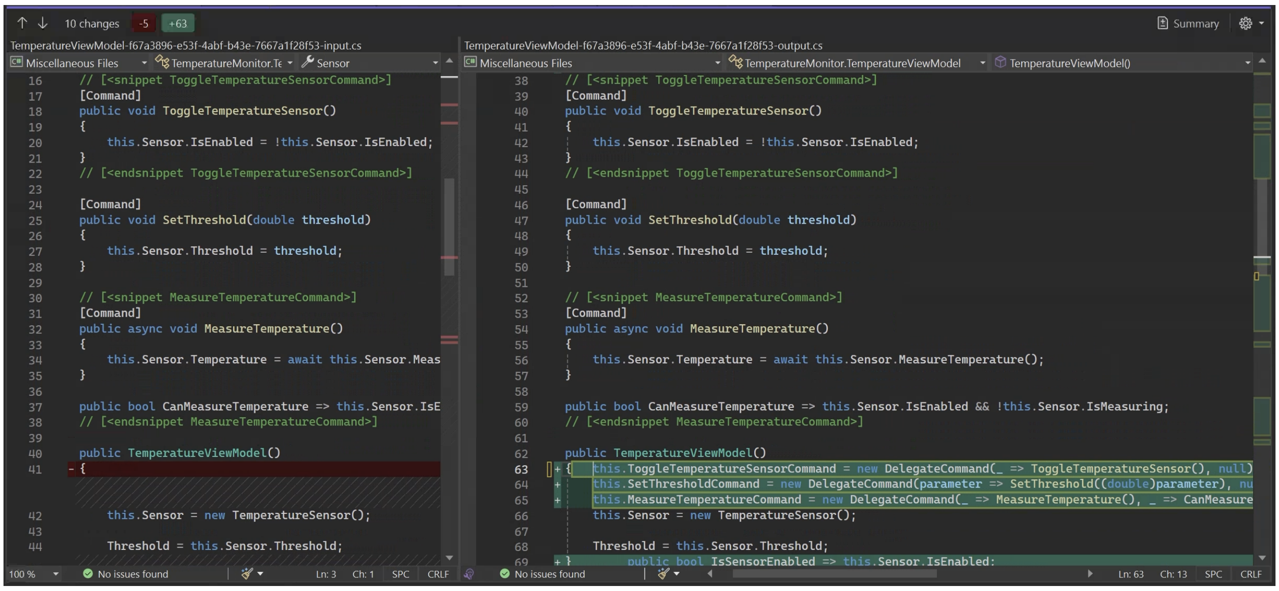
- `Is` + method name + `Enabled` (e.g. `IsToggleTemperatureSensorEnabled`)

In the following example, we have defined the `MeasureTemperature` method with the attribute `[Command]`, which will generate the command to measure the temperature using the sensor and also create the `CanExecute` method. The latter will be responsible for enabling or disabling the button based on the sensor's state to prevent repeated requests to the sensor.

```
[Command]
public void MeasureTemperature()
{
    this.Sensor.Temperature = this.Sensor.MeasureTemperature();
}

public bool CanMeasureTemperature => this.Sensor is { IsEnabled: true,
IsMeasuring: false };
```

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).



TemperatureViewModel-f67a3896-e53f-4abf-b43e-7667a1f28f53-input.cs

```
// [<snippet ToggleTemperatureSensorCommand>]
[Command]
public void ToggleTemperatureSensor()
{
    this.Sensor.IsEnabled = !this.Sensor.IsEnabled;
}
// [<endsnippet ToggleTemperatureSensorCommand>]

[Command]
public void SetThreshold(double threshold)
{
    this.Sensor.Threshold = threshold;
}

// [<snippet MeasureTemperatureCommand>]
[Command]
public async void MeasureTemperature()
{
    this.Sensor.Temperature = await this.Sensor.Meas
}

public bool CanMeasureTemperature => this.Sensor.IsE
// [<endsnippet MeasureTemperatureCommand>]

public TemperatureViewModel()
- {


    this.Sensor = new TemperatureSensor();

    Threshold = this.Sensor.Threshold;
```

TemperatureViewModel-f67a3896-e53f-4abf-b43e-7667a1f28f53-output.cs

```
// [<snippet ToggleTemperatureSensorCommand>]
[Command]
public void ToggleTemperatureSensor()
{
    this.Sensor.IsEnabled = !this.Sensor.IsEnabled;
}
// [<endsnippet ToggleTemperatureSensorCommand>]

[Command]
public void SetThreshold(double threshold)
{
    this.Sensor.Threshold = threshold;
}

// [<snippet MeasureTemperatureCommand>]
[Command]
public async void MeasureTemperature()
{
    this.Sensor.Temperature = await this.Sensor.MeasureTemperature();
}

public bool CanMeasureTemperature => this.Sensor.IsEnabled && !this.Sensor.IsMeasuring;
// [<endsnippet MeasureTemperatureCommand>]

public TemperatureViewModel()
+ { this.ToggleTemperatureSensorCommand = new DelegateCommand(_ => ToggleTemperatureSensor(), null)
+     this.SetThresholdCommand = new DelegateCommand(parameter => SetThreshold((double)parameter), nu
+     this.MeasureTemperatureCommand = new DelegateCommand(_ => MeasureTemperature(), _ => CanMeasure
    this.Sensor = new TemperatureSensor();

    Threshold = this.Sensor.Threshold;
+ }     public bool IsSensorEnabled => this.Sensor.IsEnabled;
```

Supported Scenarios

The `[Command]` aspect supports these scenarios (all covered in the Temperature Monitor example):

- **Simple command**: A method that doesn't take any parameters.
- **Command with parameters**: A method that takes a parameter.

Metalama 2025.0 adds the following features:

- **Async commands**: Commands whose implementation returns `Task`.
- **Background commands**: Commands implemented by a non-`Task` method.

All scenarios are supported following the same idiomatic pattern we have seen in the examples above.

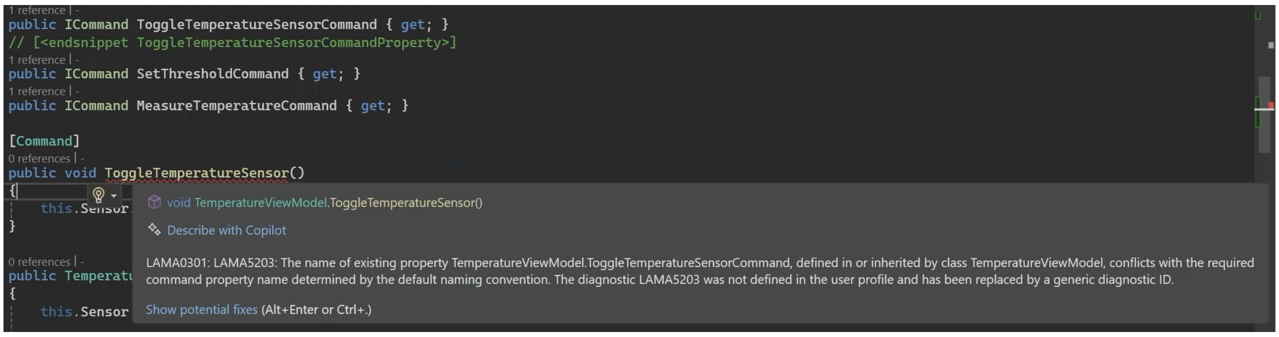
Is Metalama’s [Command] that good?

From our experience, we can say that using Metalama’s `[Command]` aspect is better than implementing the `ICommand` interface manually for the following reasons:

Benefit 1. Fewer human errors

When you manually implement the `ICommand` interface, you often end up writing repetitive code. This involves creating separate classes for each command you want to execute, even though they might only have slight differences. With Metalama, you only need to apply the `[Command]` attribute to the method, and the aspect will generate the boilerplate code for you.

And even in those cases when the automatic generation of boilerplate code conflicts with manually written code, Metalama’s `[Command]` will be safe to use. e.g., if you have a method named `ToggleTemperatureSensorCommand` in your model and you apply the `[Command]` attribute to `ToggleTemperatureSensor` property, you’ll receive an error message. This occurs because the automatically generated code will conflict with your existing code.



```
1 reference | -
public ICommand ToggleTemperatureSensorCommand { get; }
// [<endsnippet ToggleTemperatureSensorCommandProperty>]
1 reference | -
public ICommand SetThresholdCommand { get; }
1 reference | -
public ICommand MeasureTemperatureCommand { get; }

[Command]
0 references | -
public void ToggleTemperatureSensor()
{
    this.Sensor
}

0 references | -
public Temperatu
{
    this.Sensor
```

void TemperatureViewModel.ToggleTemperatureSensor()

Describe with Copilot

LAMA0301: LAMA5203: The name of existing property TemperatureViewModel.ToggleTemperatureSensorCommand, defined in or inherited by class TemperatureViewModel, conflicts with the required command property name determined by the default naming convention. The diagnostic LAMA5203 was not defined in the user profile and has been replaced by a generic diagnostic ID.

Show potential fixes (Alt+Enter or Ctrl+.)

Benefit 2. Reduce complexity

When you manually implement the `ICommand` interface, you might end up writing more code than needed. For instance, with the `ToggleTemperatureSensorCommand`, all that’s required is implementing the `Execute` method. However, we ended up creating an entire new class, passing the sensor through the constructor, and implementing a `CanExecute` method that always returns `true`, even though it’s not necessary.

This extra complexity has some drawbacks:

- It can result in an unnecessary explosion of classes in your project, making it more difficult to maintain and understand.
- It can cause scalability issues when you need to add new commands to your application.
- It may lead to code duplication, as the `Command` classes are very similar to each other.

Integration with the Observable aspect

What if I told you that there is a way to further reduce the boilerplate code in the temperature monitor example?

Metalama's `[Command]` aspect excels by generating boilerplate code for the `ICommand` interface. However, if you're looking to take it a step further, you can pair it with the `[Observable]` aspect we introduced in this article to automatically create the `INotifyPropertyChanged` interface.

Let's use the `TemperatureViewModel` class as an illustration. This class implements the `INotifyPropertyChanged` interface to inform the UI when a property is updated.

How does this relate to the `ICommand` interface? When you toggle the checkbox to activate or deactivate the sensor, the `ToggleTemperatureSensorCommand` changes the sensor `IsEnabled` property value. However, the GroupBox that allows user interaction with the sensor remains unaware of this change because it's linked to the `IsSensorEnabled` property of the `TemperatureViewModel` class. Beside the fact this property is directly related to the `Sensor` instance (which was updated by the `ToggleTemperatureSensorCommand`), no one told the UI the need to read back that value. To do it we must subscribe to the sensor's `PropertyChanged` event (added by the imlementation of the `INotifyPropertyChanged` interface) and manually trigger the `OnPropertyChanged` of the `TemperatureViewModel` class.

```
public TemperatureSensor Sensor
{
    get => this._sensor;

    set
    {
        if ( !ReferenceEquals( value, this._sensor ) )
        {
            this.UnsubscribeFromSensor();
            this._sensor = value;
            this.OnPropertyChanged( nameof(this.Sensor) );
            this.SubscribeToSensor();
        }
    }
}

public bool IsSensorEnabled => this.Sensor.IsEnabled;
```

You can see the `HandleSensorPropertyChanged` method that triggers the `OnPropertyChanged` according to the property that has changed. Note that triggering is done manually.

```
private void SubscribeToSensor()
{
    if ( this._sensor != null )
    {
        this._sensor.PropertyChanged += this.HandleSensorPropertyChanged;
    }
}

private void HandleSensorPropertyChanged( object? sender, PropertyChangedEventArgs
e )
{
    {
        var propertyName = e.PropertyName;

        if ( propertyName is nameof(this.Sensor.IsEnabled) )
        {
            this.OnPropertyChanged( nameof(this.IsSensorEnabled) );
        }

        if ( propertyName is nameof(this.Sensor.Temperature) or
nameof(this.Sensor.Threshold) )
        {
            this.OnPropertyChanged( nameof(this.CurrentTemperature) );
            this.OnPropertyChanged( nameof(this.TemperatureStatus) );
        }
    }
}
```

This is a lot of boilerplate code that can be avoided using the `[Observable]` aspect. This attribute will automatically generate the `INotifyPropertyChanged` interface and handle all notifications between properties for you. Just watch how the `TemperatureViewModel` class looks like when the `[Command]` and the `[Observable]` aspect are applied:

```
[Observable]
public partial class TemperatureViewModel
{
    public TemperatureSensor Sensor { get; set; }

    [Command]
    public void ToggleTemperatureSensor()
    {
        this.Sensor.IsEnabled = !this.Sensor.IsEnabled;
    }


    [Command]
    public void SetThreshold( double threshold )
    {
        this.Sensor.Threshold = threshold;
    }

    [Command]
    public void MeasureTemperature()
    {
        this.Sensor.Temperature = this.Sensor.MeasureTemperature();
    }

    public bool CanMeasureTemperature => this.Sensor is { IsEnabled: true,
IsMeasuring: false };

    public TemperatureViewModel()
    {
        this.Sensor = new TemperatureSensor();

        this.Threshold = this.Sensor.Threshold;
    }

    public bool IsSensorEnabled => this.Sensor.IsEnabled;

    public double Threshold { get; set; }

    public double CurrentTemperature => this.Sensor.Temperature;

    public string TemperatureStatus
    {
        get
            => this.Sensor.Temperature > this.Sensor.Threshold
                ? "Temperature is above threshold!"
                : "Temperature is below threshold.";
    }
}
```

For more details on the `[Observable]` pattern, its benefits and how to use it, please read our Implement INotifyPropertyChanged with Metalama article or the reference documentation.

Summary

In this article, we demonstrated the benefits of utilizing Metalama's `[Command]` and `[Observable]` aspects to enhance the development process of a temperature monitor application. By comparing the manual implementation of the `ICommand` interface with Metalama's automated method, we emphasized how repetitive coding and potential errors can be reduced. The traditional approach requires extensive boilerplate code for each command, increasing complexity and maintenance difficulties. In contrast, Metalama simplifies this by automatically generating essential code components, minimizing human error and improving scalability. Furthermore, by using the `[Observable]` aspect alongside `[Command]`, developers can further eliminate redundant code related to property change notifications. Ultimately, these aspects not only boost development efficiency but also enhance code readability and maintainability, providing a robust solution for developers working with common patterns in .NET applications.

This article was first published on a https://blog.postsharp.net under the title Implement ICommand with Metalama.
Discover Metalama, the leading code generation and validation toolkit for C#

- **Write and maintain less code** by eliminating boilerplate, generating it dynamically during compilation, typically reducing code lines and bugs by 15%.
- **Validate your codebase against your own rules in real-time** to enforce adherence to your architecture, patterns, and conventions. No need to wait for code reviews.
- **Excel with large, complex, or old codebases.** Metalama does not require you to change your architecture. Beyond getting started, it's at scale that it really shines.

Discover Metalama Free Edition

Related articles

- Implement INotifyPropertyChanged with Metalama
- Implementing WPF dependency properties with Metalama
- 4 Ways to Implement ICommand
- More from the Timeless .NET Engineer series