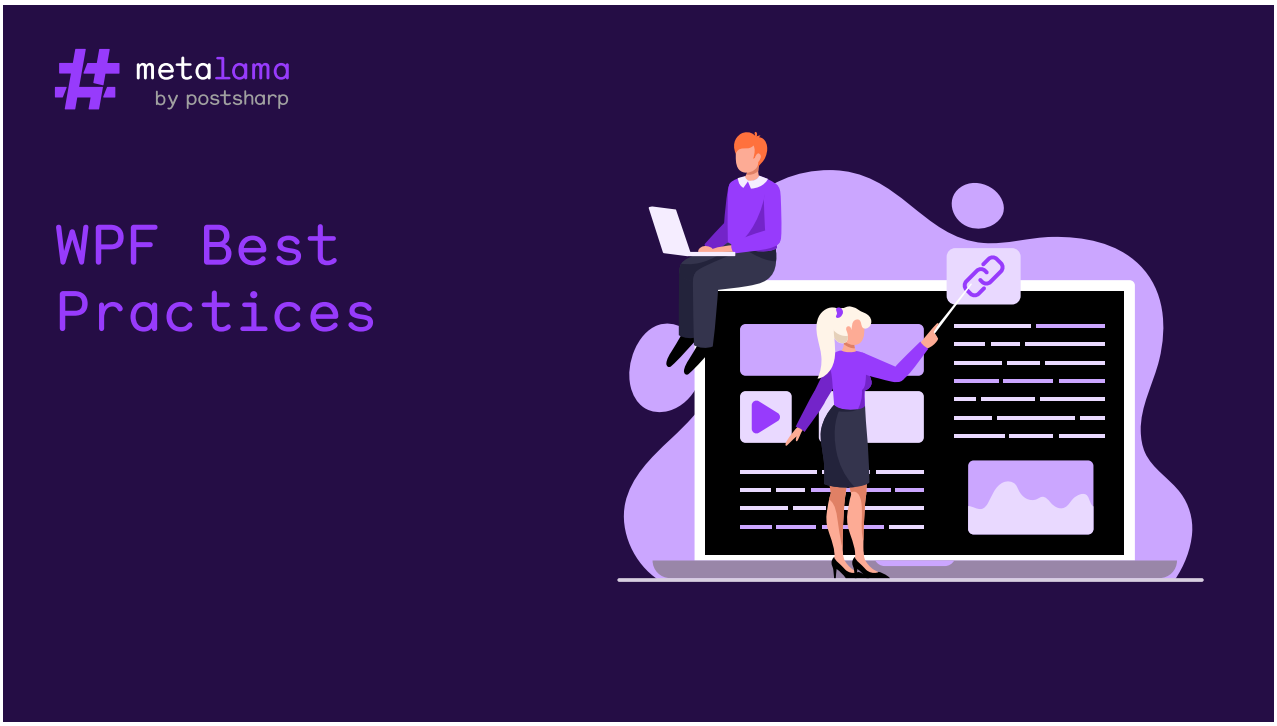
10 WPF Best Practices [2024]

blog.postsharp.net/wpf-best-practices-2024

Metalama Team November 19, 2024



Windows Presentation Foundation, or WPF, remains a vital technology for building visually appealing desktop applications on Windows. In this article, we'll explore some essential best practices for creating high-performance and maintainable WPF applications with minimal effort – updated with new tools and features released in 2024. We'll cover strategies like using the MVVM pattern, employing commands for user interactions, optimizing the complexity of the visual tree, and utilizing code generation tools.

What is WPF?

Windows Presentation Foundation is a graphical system created by Microsoft for designing user interfaces in Windows applications. It was first introduced with the .NET Framework 3.0. WPF offers a consistent way to build applications by keeping the user interface separate from the business logic. It uses a language called XAML, which stands for eXtensible Application Markup Language, to define UI elements. This approach allows designers and developers to collaborate effectively.

Moreover, WPF comes with a range of features, including data binding, 2D and 3D graphics, animations, styles, templates, and media services. These features make WPF a powerful platform for developing rich desktop client applications with visually impressive interfaces.

What are WPF applications used for?

WPF is one of the most popular UI frameworks for building desktop applications for Windows. Its principal strengths include high adaptability and performance. Its main weakness is being Windows-specific.

Since WPF is shipped with Windows as a part of the .NET Framework, it is possible to build very applications with a minimal download overhead. For those looking for a more modern development toolset, WPF has been ported to .NET Core.

WPF continues to be actively maintained and improved and remains compatible with the latest Windows versions. Although it may not receive as much attention as some newer technologies, WPF still holds its ground as a preferred choice for developers building native Windows desktop applications. This is especially true in industries where the highest levels of performance and reliability are required.

In addition to these strengths, there are exciting developments within the WPF ecosystem that promise even greater potential. For example, Avalonia XPF extends the lifespan of existing WPF applications by enabling cross-platform deployment to macOS and Linux. There is also potential for future expansion to iOS, Android, and web browsers.

As WPF has been around for almost two decades, some best practices have emerged and are still valid today.

Practice 1. Never use UI elements as your primary data storage

When developing applications, it's crucial to adopt a mindset that separates the user interface from the underlying data logic. This is a common pitfall that can lead to code that's hard to maintain and test.

One way to achieve this separation is by imagining your application running smoothly without a user interface. Focus on the fundamental data structures and operations that power its functionality. By doing so, developers are encouraged to create robust model classes that store and manage data independently of UI components. For instance, instead of storing temporary values in UI controls like text boxes, it's better to use backing variables within model classes. By linking these variables to UI elements, you establish a clear separation of concerns. This reduces fragility and maintenance challenges in the code, making your application more scalable and easier to test.

This principle also applies to complex UI structures such as tree views or stateful dependent controls like a master-detail view. In these cases, it's far better to manipulate the underlying data models and leverage WPF's data binding principles. By doing so, you have **a single "source of truth"**; any changes made are automatically reflected in the UI. This promotes a more maintainable and elegant design. This approach not only simplifies your code but also aligns with best practices for modern software development by ensuring your application's logic is cleanly abstracted from its presentation layer.

With WPF, the best design pattern for separating UI from data storage is MVVM.

Practice 2. Use MVVM - the best architecture for WPF applications

MVVM (Model-View-ViewModel), in its most basic form, is about keeping the data (**Model**) separate from the interface (**View**) by using a mediator (**ViewModel**) between them. Instead of having all the logic directly linked to the UI controls in the code file, MVVM separates the logic from the UI, making the code more modular, easier to maintain, and easier to test.

Ok, you may have heard that a lot of times. Let's see what it means with an example:

```
// Model - Just holds data
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

// ViewModel - Connects View and Model
public class MainViewModel : INotifyPropertyChanged
{
    private Person _person = new Person();

    public string Name
    {
        get { return _person.Name; }

        set
        {
            _person.Name = value;
            OnPropertyChanged(nameof(Name));
            OnPropertyChanged(nameof(Caption));
        }
    }

    public string Caption => $"{Name} ({_person.Age})";

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

The code above shows a simple example of a Model and ViewModel in WPF.

- The `Person` class is the Model, which holds the data.
- The `MainViewModel` class is the ViewModel, which connects the View (UI) to the Model. The ViewModel implements the `INotifyPropertyChanged` interface to notify the View of any changes in the data. Implementing the INotifyPropertyChanged interface can require a lot of redundant code. We'll see below how to use a code generation tool to automate the INotifyPropertyChanged boilerplate.

- The XAML code below shows the View, which is the UI of the application. The `TextBox` and `TextBlock` elements are bound to the `Name` property of the ViewModel using data binding. This allows the UI to display and update the data in the ViewModel.

```
<!-- View - Just shows the interface -->
<Window x:Class="WpfApp.MainWindow">
    <StackPanel>
        <TextBox Text="{Binding Caption}" />
    </StackPanel>
</Window>
```

This separation of concerns makes it easy to maintain each part without affecting the others. For example, if you need to change the way the data is displayed, you can modify the View without touching the ViewModel or the Model. Additionally, the model—and more importantly, the view model—can be tested independently of the user interface. This offers a significant advantage when writing unit tests.

It's a good practice to add computed properties to the ViewModel, like `Caption` in this example. The ViewModel can also contain WPF-specific properties, for instance dynamically computing the `Visibility` of an item. The objective is to make the XAML code simple. Ideally, it should be editable by non-programmers!

A problem of exposing WPF-specific properties to your ViewModel is that you cannot reuse it when porting the UI to a different platform. If this matters in your project, you can create two ViewModel layers in your application: one platform-specific, one platform-neutral. An alternative is to use a multi-platform UI framework, for instance, MAUI, Uno Framework, or Avalonia.

For more regarding the MVVM pattern, see read 4 Ways to Implement INotifyPropertyChanged on this blog.

Practice 3. Use WPF Commands to manage user interactions

Let's continue our discussion on maintaining a clear separation between UI logic and business logic in WPF applications. Now, we'll focus on using WPF Commands to handle user interactions.

Commands offer a more structured and maintainable approach compared to traditional event handlers. They help achieve this separation by allowing actions to be bound directly to properties in the ViewModel. Another benefit of the commands is that it defines a unified mechanism to bind to the `IsEnabled` state of a button or menu item. This approach results in cleaner code because it ensures that the View is solely responsible for displaying data, while the ViewModel takes care of handling interactions and enforcing business rules.

This architecture is crucial for following the MVVM pattern, as it allows the ViewModel to manage application behavior in response to user inputs without being directly tied to UI elements.

Let's take our previous code example and incorporate a command into the ViewModel. This command will handle a button click event, serving as an illustration of how commands function in practice.

```
public class SaveNameCommand : ICommand
{
    private readonly MainViewModel _viewModel;

    public SaveNameCommand(MainViewModel viewModel)
    {
        _viewModel = viewModel;
    }

    public event EventHandler CanExecuteChanged;

    public bool CanExecute(object parameter)
    {
        return !string.IsNullOrEmpty(_viewModel.Name);
    }

    public void Execute(object parameter)
    {
        // Save the data
    }
}

public class MainViewModel : INotifyPropertyChanged
{
    // Properties and methods from previous example
    // (...)

    public ICommand SaveNameCommand { get; }

    public MainViewModel()
    {
        SaveNameCommand = new SaveNameCommand(this);
    }
}

<Button Content="Save" Command="{Binding SaveNameCommand}" />
```

As you can see in this example, keeping the command separate from the UI logic:

- Allows better testability because you can easily write unit tests to verify the command's behavior without needing to interact with the UI.
- Makes the code more maintainable as the command's logic is encapsulated in a separate class, making it easier to understand and modify.

Commands also include the `CanExecute` method, which determines whether a command can be executed at any given time. By binding this logic to UI controls, you can dynamically enable or disable them based on the application's state. This feature enhances the user experience by preventing invalid actions and guiding users through valid workflows. For example, a "Submit" button can remain disabled until all required form fields are completed, reinforcing data integrity and application stability.

One drawback of using the Command pattern is that it requires a lot of boilerplate code. However, several approaches can help you reduce redundant code with WPF commands. One of them is Metalama, a code generation and validation framework for C#, but you can also use Microsoft's MVVM Toolkit or ReactiveUI.

For more about WPF commands, read 4 Ways to Implement ICommand on this blog.

Practice 4. Use dependency properties in your user controls

Dependency properties are a powerful feature in WPF that enable data binding, styling, and animation. They allow you to extend the functionality of existing controls and create custom controls with properties that can be set in XAML.

When creating custom user controls, it's essential to use dependency properties for properties that need to be data-bound, styled, or animated. By defining dependency properties, you can take advantage of WPF's built-in features and ensure that your controls integrate seamlessly with the rest of the framework.

Let's say you want to create a custom control that includes a label and a textbox. You want to be able to set the label text and the textbox text from the XAML. You can do that by creating dependency properties for the label and textbox text:

```
public class CustomControl : Control
{
    public static readonly DependencyProperty LabelTextProperty =
DependencyProperty.Register(
        "LabelText", typeof(string), typeof(CustomControl), new
PropertyMetadata(default(string)));

    public string LabelText
    {
        get => (string) GetValue(LabelTextProperty);
        set => SetValue(LabelTextProperty, value);
    }

    public static readonly DependencyProperty TextBoxTextProperty =
DependencyProperty.Register(
        "TextBoxText", typeof(string), typeof(CustomControl), new
PropertyMetadata(default(string)));

    public string TextBoxText
    {
        get => (string) GetValue(TextBoxTextProperty);
        set => SetValue(TextBoxTextProperty, value);
    }
}
```

Now, to use it in your WPF application, you can set the `LabelText` and `TextBoxText` properties in the XAML like this:

```
<local:CustomControl LabelText="Name:" TextBoxText="{Binding Name}" />
```

Dependency properties inherently support change notification, so the WPF framework knows when the property value changes and can respond accordingly. This allows WPF to re-render UI elements or trigger property-specific logic when a property is updated (without having to implement the `INotifyPropertyChanged` interface).

As with `INotifyPropertyChanged` and commands, the downside of dependency properties is that they require some redundant code. One way to reduce it is to use Metalama or Roslyn source generators.

For more about WPF dependency properties, read Implementing custom dependency properties in WPF (+example) on this blog.

Practice 5. Use an on-the-fly code generation tool

As you can see from the prior sections, many WPF best practices involve writing a lot of boilerplate code. Fortunately, in 2024, most of this repetitive code can be generated on the fly.

Benefits

When it comes to WPF applications, code generation tools can be particularly beneficial for:

1. **Writing less code**: By speeding up initial development phases, these tools and practices allow developers to focus more on core business logic rather than getting bogged down in routine coding tasks. This productivity boost can lead to faster project delivery and reduced time-to-market.

2. **Maintaining less code**: Writing code is only about 30% of its total cost of ownership; for most enterprise projects, maintenance represents the majority of the cost. Since every line of code has a cost, it's important to keep the count low. Avoid tools that generate code you need to check into your source repository because this code, even if it's generated, still needs to be maintained. Instead, use a tool that generates code on the fly during the build.

3. **Ensuring consistency across the whole codebase**: Code generation tools ensure a uniform code structure throughout the project, which helps maintain coding standards and best practices across different components and teams. This uniformity reduces variations in implementation that can lead to discrepancies or integration issues later on, fostering a more cohesive and maintainable code environment.

Available tools

- Metalama allows you to automate repetitive tasks in your codebase using aspects, special custom attributes that execute within the compiler or the IDE and dynamically transform your source code. Metalama provides a set of predefined aspects that can be used to automate common tasks in WPF applications, such as implementing the INotifyPropertyChanged interface, dependency properties, and WPF commands.
- MVVM Community Toolkit includes some Roslyn code generators for commands and `INotifyPropertyChanged`. It does not come with as many features as Metalama, but it's signed by Microsoft!
- Fody is an MSIL code weaving tool. It can be used to automatically implement `INotifyPropertyChanged` in WPF applications thanks to the PropertyChanged.Fody plug-in, which also comes with its Roslyn code generator.

Of course, you can always write your own Roslyn code generator.

Practice 6. Minimize visual tree complexity

The visual tree in WPF is the hierarchy of UI elements rendered on the screen, and each element contributes to processing such as layout recalculation, rendering, and event handling. A complex or deeply nested visual tree means more work for the rendering engine, which can result in slower performance, increased memory usage, and longer UI response times.

Why this is important:

- **Rendering Performance**: Each element in the visual tree needs to be rendered and laid out. A deeply nested tree requires more processing, which can lead to slower rendering and layout recalculations, especially when the UI changes dynamically (e.g., resizing windows or updating data).
- **Memory Usage**: Each UI element consumes memory. An overly complex visual tree with unnecessary elements can quickly consume resources, leading to higher memory usage and potential performance bottlenecks.
- **Event Handling Overhead**: WPF handles routed events by traversing the visual tree. The more elements there are, the more time it takes for events to be processed, which can affect UI responsiveness.

Let's take the following code as a simple example of this:

```
<StackPanel>
    <Grid>
        <StackPanel Margin="10">
            <Button Content="Click Me" />
        </StackPanel>
    </Grid>
</StackPanel>
```

In this example, the `Button` is nested inside a `StackPanel` inside a `Grid` inside another `StackPanel`. This creates a visual tree with multiple layers of nesting, which can impact performance. To optimize the visual tree, you can simplify the structure by removing unnecessary containers:

```
<Button Content="Click Me" Margin="10" VerticalAlignment="Top" />
```

By flattening the visual tree and reducing unnecessary nesting, you can improve rendering performance, reduce memory usage, and enhance the overall responsiveness of your WPF application.

Practice 7. Use virtualization for long lists

Virtualization means that only the UI elements currently visible in the viewport are created and rendered, while off-screen elements are not generated until they are scrolled into view.

Why Virtualization Is Important:

- **Memory Efficiency**: Without virtualization, all items in a list or collection are generated and kept in memory, even if they are not visible. This can lead to high memory consumption, especially for large data sets.
- **Improved Rendering Performance**: Virtualization reduces the number of elements that the WPF rendering engine needs to measure, arrange, and paint. This results in faster scrolling and smoother user interactions. ```
- **Reduced Load Time**: The application starts faster because it doesn't need to create the entire list at once.

Here's an example of a ListBox bound to a `LargeItemCollection` containing 1,000 items with virtualization enabled:

```
<ListBox ItemsSource="{Binding LargeItemCollection}"
         VirtualizingStackPanel.IsVirtualizing="True"
         VirtualizingStackPanel.VirtualizationMode="Recycling" />
```

In this example, setting `IsVirtualizing` to `True` enables virtualization for the `ListBox`. The `VirtualizationMode` is set to `Recycling`, meaning that item containers are reused as they scroll out of and back into view. This improves performance by reducing the need to create new UI elements. WPF has built-in support for virtualization through the `VirtualizingStackPanel`, which is the default panel for controls like `ListBox`, `ListView`, and `DataGrid`. So, next time you use one of these controls, remember to enable virtualization to optimize performance and memory usage.

Practice 8. Use async or background methods for long operations

When a long-running operation (e.g., file I/O, database queries, or web service calls) is executed on the same thread as the application's UI, it will block the UI and cause the application to become unresponsive until the operation completes. To prevent this and keep the user interface (UI) responsive, execute long-running operations asynchronously or on a background thread.

By using `async` methods or running long operations in the background using `Task.Run`, you will:

- **Improve UI responsiveness**: By executing long operations asynchronously or on a background thread, the UI remains responsive and can continue to handle user interactions while the operation is in progress.
- **Prevent UI freezes**: Blocking the UI thread can lead to a poor user experience, as the application appears frozen or unresponsive. Using asynchronous or background methods prevents this issue and maintains a smooth user experience.
- **Improve overall performance**: Asynchronous operations can improve the overall performance of the application by allowing multiple tasks to run concurrently. This can lead to faster execution times and better resource utilization.

Here's an example of a long-running operation that reads data from a file and updates the UI. If done incorrectly, it will block the UI thread and make the application unresponsive:

```
public class MainViewModel
{
    public void LoadData()
    {
        // This blocks the UI thread for 3 seconds
        // User can't click buttons, scroll, or see updates
        Thread.Sleep(3000);  // Simulating heavy work
        MessageBox.Show("Done!");
    }
}
```

To make this operation asynchronous, you can use async/await in C#:

```
public class MainViewModel : INotifyPropertyChanged
{
    private bool _isLoading;
    public bool IsLoading
    {
        get => _isLoading;
        set
        {
            _isLoading = value;
            OnPropertyChanged(nameof(IsLoading));
        }
    }

    public async Task LoadDataAsync()
    {
        IsLoading = true;
        // UI remains responsive while this runs
        await Task.Delay(3000);  // Simulating heavy work
        IsLoading = false;
        MessageBox.Show("Done!");
    }
}
```

In this example, the `LoadDataAsync` method is marked as `async`, and the heavy work is done inside the `Task.Delay` method. By using `await`, the method returns control to the UI thread while the operation is in progress, allowing the UI to remain responsive. The `IsLoading` property is used to indicate that the operation is in progress, which can be bound to a loading indicator in the UI. The same behavior can be achieved by using `Task.Run` to execute the long-running operation on a background thread:

```
public async Task LoadDataAsync()
{
    IsLoading = true;
    await Task.Run(() =>
    {
        // This runs on a background thread
        Thread.Sleep(3000);  // Simulating heavy work
    });
    IsLoading = false;
    MessageBox.Show("Done!");
}
```

By offloading these tasks to background threads or running them asynchronously, you prevent the UI from freezing, enhancing the overall user experience and application performance.

Practice 9. Minimize code-behind

This best practice is closely related to the MVVM pattern and the separation of concerns explained earlier. But first, let’s clarify what code-behind is in WPF.

The code-behind is where event handlers, initializations, and some logic related to the UI are traditionally placed. While the MVVM (Model-View-ViewModel) pattern encourages minimizing code-behind, there are still legitimate use cases where its presence is reasonable or necessary:

- **Handling lifecycle events**: Code-behind is used to handle lifecycle events of the window or control, such as `Loaded`, `Closing`, or `SizeChanged`.
- **Simple UI logic**: Code-behind is used for simple UI logic that doesn't fit well in the ViewModel, such as setting initial values or handling UI-specific behavior.
- **Animations and storyboards**: Code-behind is used to create animations and storyboards that are tightly coupled to the UI elements.

Although there are other legitimate use cases, their use can be kept to a minimum:

- **Event Handlers for UI Controls**: Code-behind is used to handle events raised by UI controls, such as button clicks, text changes, and mouse interactions.

  ```
  private void Button_Click(object sender, RoutedEventArgs e)
  {
      // Handle button click event
  }
  ```

  One way to minimize code-behind is to use commands in the ViewModel to handle these events, as discussed earlier.

- **UI Element Initialization**: Code-behind is used to initialize UI elements, set properties, and define the layout.

  ```
  public MainWindow()
  {
      InitializeComponent();
      // Initialize UI elements
      MyButton.Content = "Dynamically set content";
  }
  ```

  While this is a valid use of code-behind, consider using data binding and styles to set properties and define the layout in XAML, reducing the need for code-behind.

- **Direct Access to UI Components**: Code-behind is used to access and manipulate UI elements directly.

  ```
  MyTextBox.Text = "Updated text directly in code-behind";
  ```

  To minimize code-behind, consider not using UI components to store data; instead, use data binding and commands to interact with UI elements indirectly through the ViewModel.

While code-behind has its place in WPF applications, it's essential to minimize its use and follow the MVVM pattern to maintain a clean separation of concerns and improve the testability and maintainability of your code.

Practice 10. Don’t reinvent the wheel

This final advice encourages developers to leverage existing, well-tested tools, libraries, and components instead of spending valuable time and resources creating complex controls or frameworks from scratch. The WPF ecosystem offers a wide range of third-party libraries and tools that can help developers build high-quality applications more efficiently.

By sourcing an existing component instead of building your own, you will:

- **Save development time**: Custom controls and automation patterns can be time-consuming and difficult to implement. Using third-party libraries allows developers to focus on building core application features rather than low-level UI elements or libraries.
- **Improve quality**: Established vendors offer components and libraries that have been extensively tested for functionality, performance, and compatibility, ensuring that developers get robust solutions out of the box.
- **Deliver a consistent developer experience**: Professional controls and libraries often come with polished designs and flexible customization options that can improve the overall experience of coding an application.
- **Get support and benefit from the community**: Third-party libraries typically come with documentation, support forums, and community knowledge, making it easier to find solutions to common problems. Some vendors offer tools that automatically implement design patterns, such as MVVM, which help standardize code and improve productivity. We talked earlier about Metalama and its outstanding capabilities to automate repetitive tasks in your codebase using aspects. Besides allowing you to generate your own aspects (and it’s very good at it), Metalama also provides a set of predefined aspects that can be used to automate common tasks in WPF applications. The Metalama Patterns consist of an open-source library of production-ready aspects that implement the most common design patterns for C#. In the context of WPF applications, Metalama can automate the implementation of the INotifyPropertyChanged, WPF commands through the ICommand interface, dependency properties, and other common patterns.

Several vendors offer high-quality UI components like data grids or charts. You will find a comprehensive list on ComponentSource’s website.

Conclusion

Despite being almost two decades old, WPF is still an excellent and highly popular choice for building desktop Windows applications. By following best practices, such as implementing the MVVM pattern, using commands for user interactions, and taking advantage of dependency properties, you can build applications that are not only visually stunning but also easy to maintain and scale.

Utilizing techniques like virtualization and asynchronous operations ensures that applications perform optimally. Minimizing code-behind helps maintain a clear separation of concerns, which is crucial for a clean code architecture. Additionally, using code generation tools like Metalama and tapping into existing libraries allows you to streamline your workflow. This means you can focus on core functionalities rather than reinventing the wheel.

As WPF continues to evolve within the ever-changing tech landscape, adhering to these practices will ensure that your applications remain efficient, reliable, and adaptable to future technological advancements.

This article was first published on a https://blog.postsharp.net under the title 10 WPF Best Practices [2024].
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

- 4 Ways to Implement INotifyPropertyChanged
- 4 Ways to Implement ICommand
- More from the Timeless .NET Engineer series