Prism: An Open-Source Framework for Building XAML Applications in WPF for Instrumentation

Tom Roberts

[Summary] This paper examines the Prism framework as a foundational open-source solution for building large-scale, modular WPF applications in instrumentation and scientific software. It discusses the architectural challenges inherent in WPF and MVVM development and explains how Prism addresses these challenges through standardized infrastructure such as dependency injection, modularity, region-based composition, navigation, and loosely coupled communication. Key Prism features—including the ViewModel Locator, Region Manager, Event Aggregator, and Dialog Service—are explored in the context of scalability, testability, and long-term maintainability. Finally, the paper considers Prism's evolution as an open-source project and its continued relevance for enterprise-grade instrumentation applications.

1 Introduction

Open-source frameworks are essential for instrumentation and scientific software as they provide pre-built, community-vetted foundations that allow developers to skip complex infrastructure tasks and focus on specialized features. These frameworks accelerate development cycles by offering modular, reusable components for large-scale enterprise applications, ensuring they can scale efficiently to meet the demands of timely feature additions without breaking existing ones. Furthermore, the inherent transparency of open source fosters a culture of collaborative problem-solving and rigorous peer review, which enhances software reliability.

Over the past 10–15 years, there has been a shift from Windows Forms (WinForms) to Windows Presentation Foundation (WPF) for enterprise line-of-business applications. WPF offers a more modern User Interface (UI) framework with powerful data binding, Model-View-ViewModel (MVVM) support, and styling/templating, which makes large applications easier to maintain and scale. WPF has a significant learning curve because it requires mastering several advanced concepts simultaneously, including XAML[1),2)], a declarative markup language for UI design that is distinct from traditional C# code. WPF's core architectural pattern, MVVM[3)–5)], demands a major shift in thinking from simpler event-driven models like WinForms to a more decoupled approach. Additionally, understanding complex internal mechanics such as dependency properties, routed events, and data binding[6)–9)] can be daunting for beginners. Unlike older frameworks, WPF also introduces sophisticated and powerful layout systems, templates, and styles but adds significant verbosity and initial confusion. Finally, debugging can be difficult because runtime errors in the XAML object tree often produce abstract or unhelpful messages.

Implementing the MVVM pattern is essential for professional WPF development because it decouples the UI from the application's core logic, enabling applications to scale without becoming unmanageable. By moving state and behavior into a separate ViewModel, developers create logic that is unaware of the View and communicates through data

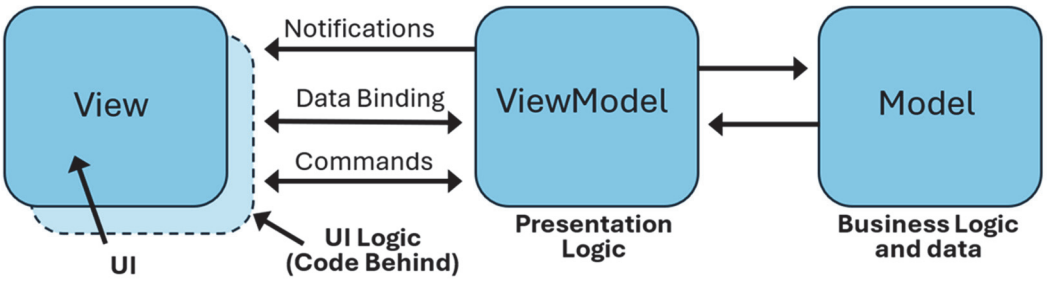

Figure 1 Relationships and interactions between MV

binding, commanding and notifications. As illustrated in Figure 1, the View and ViewModel interact indirectly through well-defined mechanisms, while the Model remains isolated from UI concerns. This separation allows designers to modify the visual experience in XAML without risk of breaking underlying functionality, while also facilitating comprehensive unit testing of business logic in complete isolation from the UI thread.

While the core WPF framework provides the raw building blocks for MVVM, such as data binding and commands, it lacks the high-level infrastructure required to handle complex application-wide concerns. Professional MVVM frameworks fill these gaps by providing standardized "plumbing" for common tasks like message-based communication between disconnected components, integrated dependency injection (DI) for managing object lifetimes, and automated property change notification. Without these frameworks, developers often resort to writing repetitive "boilerplate" code for every new screen, which increases the likelihood of human error and significantly slows development cycles.

Scaling a WPF application to an enterprise level requires a robust modular architecture that frameworks simplify through advanced navigation and composition systems. These frameworks allow large teams to build and maintain independent features as separate plug-ins that can be dynamically loaded at runtime. This modular approach ensures that a change in one functional area of the application does not inadvertently cause a failure in another, making the entire system more resilient and easier to maintain.

Ultimately, leveraging an established MVVM framework standardizes the development process and enforces architectural best practices across a project. By providing a common language and set of conventions, these tools help teams avoid the "spaghetti code" that frequently plagues long-lived desktop applications. As applications evolve, built-in support for sophisticated features—such as automatic View-to-ViewModel discovery and cross-component event systems—enables a level of agility and maintainability that is difficult to achieve through manual implementation alone.

2 Prism Library Evolution

The Prism MVVM Framework was established in 2008 by Microsoft's patterns & practices team, originally under the name Composite Application Library. It was developed to provide official guidance and a set of libraries for building "composite applications" - complex, large-scale WPF software that requires a modular and loosely coupled architecture[10),11)]. By implementing the MVVM pattern along with other architectural principles like DI and Event Aggregation, Prism was designed to solve the challenges of maintaining and testing sophisticated XAML-based applications. It allows multi-disciplinary teams to develop, test, and evolve individual application components independently, ensuring the UI remains separate from business logic to improve long-term flexibility and code re-use. In 2015, Microsoft transitioned the project to the open-source community[12),13)].

As of 2026, the Prism Library remains a dominant framework for building modular XAML applications. Its core package, Prism.Core, has surpassed 19.5 million total downloads on NuGet, with a daily average of approximately 8,600 downloads. The user base is extensive and well-established, supported by a mature ecosystem of community-driven documentation and active development targeting the latest .NET environments.

Prism[14)] fundamentally simplifies the MVVM pattern by offering built-in components which ensure a clean separation between UI and business logic. Prism's integration with common DI containers allows developers to manage object lifetimes and resolve dependencies effortlessly, promoting a loosely coupled system. This separation of concerns enhances testability, allowing ViewModels and services to be verified independently using design-time mocks or automated tests without a live UI. Modularity is a core strength, enabling large applications to be decomposed into independent modules that can be developed, tested, and updated by separate teams in parallel. The framework further supports a composite UI approach through regions and navigation services, allowing dynamic content loading into specific placeholders. Advanced communication between these decoupled parts is facilitated by the Event Aggregator, which provides a thread-safe, loosely coupled publisher/subscriber mechanism. By standardizing these architectural patterns, Prism reduces technical debt and ensures that applications remain scalable and maintainable as requirements evolve.

A trend among open-source maintainers is the shift toward tiered licensing models to address the long-term sustainability of open-source projects and prevent maintainer burnout.

In 2025, Prism migrated from a community to a dual community / commercial license model. These licensing structures are beneficial because they create a more sustainable financial model by allowing open-source maintainers to earn directly from commercial use while keeping the code accessible for community use. By balancing commercial interests with open-source principles, these frameworks can secure the funding necessary for professional-grade support, dedicated maintenance, and faster feature development. Ultimately, this trend helps stabilize the software supply chain, ensuring that the essential infrastructure upon which 96% of organizations[15] now rely remains actively supported into the future.

3 Integration of Dependency Injection (DI) containers in Prism

DI is a critical design pattern that decouples the creation of an object from its usage, facilitating a separation of concerns that is vital for scalable software. By centralizing the management of instances, developers can avoid hard-coded dependencies, making applications more modular and significantly easier to unit test through mock injections. The Prism MVVM framework streamlines this process by integrating with popular DI containers like Unity or DryIoc, providing a dedicated ViewModel Locator to automatically resolve and wire up dependencies at runtime. Prism further enhances instance management by handling object lifetimes, ensuring that services and components are properly registered as singletons or transient instances based on application needs. Without DI, developers often resort to the Singleton pattern or manual instantiation, which can lead to severe pitfalls such as tight coupling and rigid codebases that are difficult to modify. A lack of centralized management frequently causes memory leaks, as long-lived objects may hold onto references of short-lived components, preventing the garbage collector from reclaiming memory. These leaks are notoriously difficult to track because they often manifest as gradual performance degradation rather than immediate crashes. Identifying these hidden references can be extremely difficult and time consuming to track down and resolve. Prism's structured approach ensures that dependencies are explicit and lifecycle-managed, mitigating the risk of memory leaks and the extreme debugging challenges associated with manual instance control.

4 Loose coupling benefits

In software design, loose coupling refers to components interacting through well-defined interfaces rather than direct references. Building features with loose coupling in Prism offers several critical benefits:

- Independent Evolution: Features can be developed, updated, or replaced in isolation, ensuring that changes in one module do not cause cascading failures or require redeploying the entire application.
- Enhanced Testability: Because components are not tightly bound, they can be easily mocked or stubbed, allowing for more reliable unit testing in isolation.
- Improved Maintainability: With a clear separation of concerns, bugs are easier to isolate, and the codebase remains more manageable as the application scales.
- Scalability: Different teams can work on separate functional areas simultaneously without blocking each other, speeding up the overall development lifecycle.

5 Modules

A Prism module is a self-contained, feature-focused unit in a Prism application that bundles related views, ViewModels, services, and registrations. It allows developers to build and load application functionality independently, making large WPF apps easier to scale, test, maintain, and evolve over time. When an application has many modules, it is best to locate them outside the main Prism (shell) project. Typically, the shell project contains only the application startup, shared infrastructure, and layout, while each module lives in its own separate class library project. This keeps features isolated, reduces coupling, and allows teams to work on modules independently.

Figure 2 shows sections of a typical instrument application window: MenuBar, ToolBar, SideBars, StatusBar, Chart area etc. The MenuBar may have many defined Regions such as: FileMenuRegion, TraceMenuRegion, DisplayMenuRegion, etc. The ToolBar could have similar or related regions to the MenuBar regions: FileToolBarRegion, TraceToolBarRegion, DisplayToolBarRegion, etc. with shared functionality. For example, views, ViewModels, services, and registrations serving the TraceMenu and TraceToolBar regions would most likely be co-located in the same Prism Module.

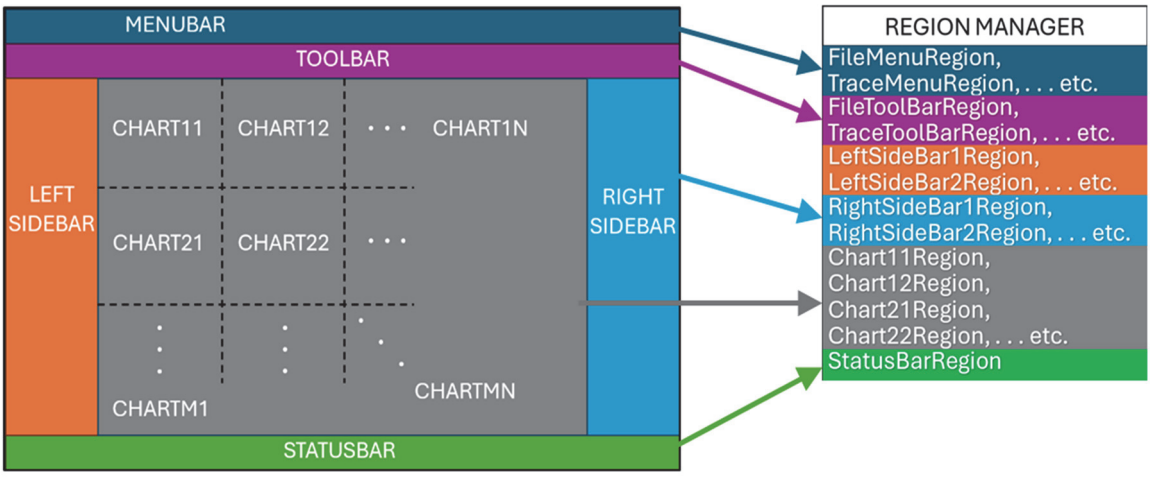

Figure 2 Application Main Window sections and Region names

6 Custom Controls for Instrumentation

Standard controls provided by WPF are usually insufficient for applications developed for instrumentation products. Interactive custom chart controls (e.g. Tabular, Rectangular, Smith, Polar, Gage, Vector Constellation, Eye Diagram, 3D surface[16], etc.) with their own properties are required. Many of these charts are available from third-party vendors but rarely satisfy all customer UI requirements so custom controls are often developed internally. Custom controls are another modular component that can be developed independently and evolve with customer requirements. These interactive custom controls can be treated like any other standard control and injected into regions as an instance or instances using Prism. The development of custom controls is not discussed in this paper, but resources are available at Pluralsight[17]. From a Prism perspective, custom controls function as modular components and are often packaged within feature modules.

7 The ViewModel Locator

When using the MVVM pattern, a way to connect to the Views `DataContext` to the instance of the View's ViewModel is necessary. Prism's `ViewModelLocator` simplifies making this connection. While several approaches exist, Prism's `ViewModelLocator` avoids manual wiring by automatically creating and assigning the ViewModel to the view's `DataContext`. Prism's implementation is lightweight and container driven. It works by setting a single attached property on the view, which triggers Prism to resolve the ViewModel using dependency injection. Prism relies on naming conventions to locate the correct ViewModel, expecting views to live in a Views namespace and ViewModels in a ViewModels namespace. The default rule appends "ViewModel" to the view name to find the corresponding ViewModel. Developers can override the naming convention or register ViewModels explicitly to fully control the mapping.

8 Regions and Region Manager

Regions are the primary mechanism to inject or display views to users throughout the application. A Region is simply a placeholder for dynamic content and can be created in XAML or in code. Regions have no knowledge of the Views that will be moved in and out of the Region. The Region Manager maintains the collection of the regions and allows access to them. A Region is defined in a Region Adapter which is host control for the Region. Prism supports various Region Adapters: `ContentControl`, `ItemsControl`, and Selector controls such as `ComboBox`, `ListBox`, `Ribbon` and `TabControl`. Prism's supported region adapters cover almost all scenarios needed; however, custom region adapters can be constructed. For example, defining a Region in a `StackPanel` would require a custom `StackPanel` Region Adapter[14]. The chart area in Figure 2 has mxn chart regions with grids hosted in a `TabControl`, for example. Any chart type instance could be moved in or out of any mxn chart regions using Region Navigation.

9 Region Navigation

Region (or View) Navigation in Prism provides a loosely coupled way to move views into and out from regions, making it ideal for MVVM scenarios where ViewModels should not reference view types directly. Region Navigation manages the full navigation lifecycle, including activating and

deactivating views and maintaining back and forward navigation stacks. Navigation is performed using URIs, which act as unique keys mapped to view types, eliminating direct dependencies on views. To enable navigation, views must be registered with Prism using the container registry, typically by calling `RegisterForNavigation`, which by default uses the view's name as the key. Once registered, navigation can be triggered through the `IRegionManager` by calling `RequestNavigate` with a region name and navigation key. Alternatively, navigation can be requested directly on an `IRegion` instance retrieved from the region manager. The region manager is usually accessed through constructor injection. This approach allows ViewModels to participate cleanly in navigation while preserving MVVM principles. Additionally, Region Navigation supports passing parameters and tracking where a view is within the navigation process.

10 Communicating between different areas of the application

Prism helps WPF applications become more decoupled, making them easier to build, maintain, extend, and test, but this decoupling can make communication between application areas more complex. One common challenge is enabling Views and ViewModels to communicate with different areas of the application without tight dependencies. Prism addresses this problem through the `IEventAggregator` interface which provides a loosely coupled, event-based communication mechanism. The Event Aggregator works using publishers and subscribers, where one part of the application publishes an event and other parts subscribe to receive it. Multiple publishers and subscribers can exist for the same event as shown in Figure 3. Subscribers must register for an event before publishers send it, and once published, the Event Aggregator routes the event to all subscribers. Events can also carry data objects, known as a payload, allowing information to be passed between application areas. Subscribers can filter events based on payload values to ensure they only handle relevant messages. The EventAggregator uses weak references by default to prevent memory leaks, but subscribers that opt into strong references must be manually unsubscribed.

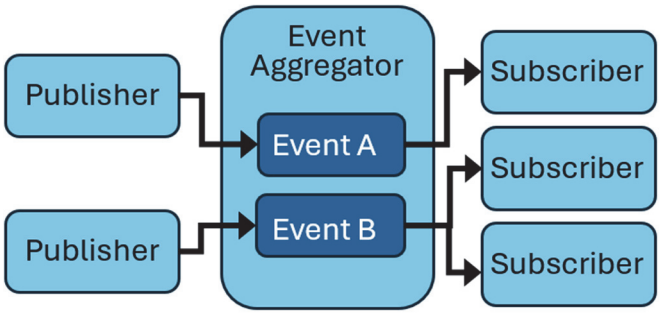

Figure 3 Event Aggregator[11)]

11 Dialogs and the Dialog Service

Applications frequently need to prompt users for input or display notifications, which is typically done through dialogs. A dialog in Prism is a `UserControl` designed to display information, request confirmation, or collect user input. In Prism, dialogs are shown without directly tying ViewModels to UI elements. Prism uses the `IDialogService`, which allows dialogs to be invoked from ViewModels without referencing views. Dialogs can be shown in either modal mode, which blocks interaction with the rest of the application, or non-modal mode, which allows continued interaction and even multiple dialog instances.

Dialogs in Prism have a ViewModel that implements the `IDialogAware` interface. This interface provides key features such as setting the dialog title, controlling whether it can close, responding when it opens or closes, and receiving parameters for initialization. Dialogs are triggered by commands in the ViewModel, keeping business logic decoupled from the UI. Dialogs must be registered with the Prism DI container. Once registered, the `IDialogService` can be used to display the dialog in a clean, testable, and maintainable way. The dialog service can also be used to show multiple shells[18)] or what is sometimes called detached windows.

12 Conclusions

This paper highlights the main features of Prism and its benefits in providing a standardized infrastructure to accelerate front-end software development for instrumentation. The Prism MVVM framework is indispensable for large-scale applications because it provides a robust, modular architecture that allows independent teams to develop, test, and deploy features simultaneously without conflict. Its core features - such as Dependency Injection, Event Aggregation, and region management - are essential for managing the high complexity of modern enterprise software by ensuring

components remain loosely coupled. Prism eliminates the need for developers to write repetitive boilerplate code or redesign foundational infrastructure. This pre-built functionality significantly reduces overall development time, allowing teams to focus exclusively on business logic rather than boilerplate plumbing. Furthermore, Prism's navigation and modularity features provide a standardized "common shell" that ensures consistent UI behavior across large codebases. Ultimately, using this established framework future-proofs applications by making them easier to maintain and evolve as business requirements grow over time.

13 References

1) Huber, T. C. (2024, June). XAML layout in depth [Video]. Pluralsight. www.pluralsight.com
2) Nathan, A. (2014). XAML unleashed. Sams Publishing.
3) Noyes, B. (2021, October). WPF MVVM in depth [Video]. Pluralsight. https://www.pluralsight.com/courses/wpf-mvvm-in-depth
4) Huber, T. C. (2022, October). WPF and MVVM: Advanced model treatment [Video]. Pluralsight. www.pluralsight.com
5) Huber, T. C. (2022, October). WPF and MVVM: Test driven development of ViewModels [Video]. Pluralsight. https://www.pluralsight.com/courses/wpf-mvvm-test-driven-development-viewmodels
6) MacDonald, M. (2012). Pro WPF 4.5 in C#: Windows Presentation Foundation in .NET 4.5 (4th ed.). Apress. doi.org
7) Noyes, B. (2022, October). WPF Data Binding in Depth [Video]. Pluralsight. https://www.pluralsight.com/courses/wpf-data-binding-in-depth
8) Noyes, B. (2022, October). Extending XAML applications with behaviors [Video]. Pluralsight. https://www.pluralsight.com/courses/extending-xaml-applications-behaviors
9) Noyes, B. (2022, October). WPF Productivity Playbook [Video]. Pluralsight. https://www.pluralsight.com/courses/wpf-productivity-playbook
10) Brumfield, B., Cox, G., Hill, D., Noyes, B., Puleio, M., & Shifflett, K. (2011). Developer's guide to Microsoft Prism 4: Building modular MVVM applications with Windows Presentation Foundation and Microsoft Silverlight. Microsoft Press.
11) Microsoft Patterns & Practices. (2014, April). Developer's guide to Microsoft Prism Library 5.0 for WPF, Microsoft Press. http://download.microsoft.com/download/5/2/6/52627B2B-AE05-4781-AA5D-7EBCC4C86D9B/Prism5forWPF.pdf
12) Prism Library. (2024). Prism (Version 9.0) [Computer software]. GitHub. https://github.com/PrismLibrary/Prism
13) Prism Library. (2024, February). Prism Library [Website]. https://prismlibrary.com/
14) Lagunas, B. (2022, May). Introduction to Prism 7 for WPF [Video]. Pluralsight. https://www.pluralsight.com/courses/prism-wpf-introduction
15) OpenLogic by Perforce, Eclipse Foundation, & Open Source Initiative. (2025). 2025 State of Open Source Report. https://www.openlogic.com/resources/state-of-open-source-report
16) Xu, J. (2016). Practical .NET chart development and applications: Advanced chart programming for real-world .NET 4.5 applications using C#, WPF, and MVVM. UniCAD Publishing.
17) Lagunas, B. (2024). Introduction to WPF custom controls [Video]. Pluralsight. https://www.pluralsight.com/courses/wpf-custom-controls-introduction
18) Lagunas, B. (2021, July). Prism problems & solutions: Showing multiple shells [Video]. Pluralsight. https://www.pluralsight.com/courses/prism-showing-multiple-shells

Author

Tom Roberts
Service Infrastructure Solutions
US Division
Test & Measurement Company

Publicly available