Prism helps you to design and build applications using loosely coupled components that can evolve independently。these applications are "built to last" and "built for change." These types of applications are known as composite applications.

Simple applications that do not have these requirements may not benefit from the Composite Application Library

Such complex applications typically feature multiple screens, rich user interaction, a consistent appearance, and role-determined behavior

Additionally these complex applications typically use layered architectures that may be physically deployed across tiers, strong separation of concerns, and loosely coupled components

**Why Use Prism?**

## Client Application Development Challenges

Application requirements can change over time，It requires an architecture that allows individual parts of the application to be independently developed and tested and that can be modified or updated later, in isolation, without affecting the rest of the application

It can be a significant challenge to decide how to design the application so that multiple developers or subteams can work effectively on different pieces of the application independently,

Designing and building applications in in which the components are very tightly coupled can lead to an application that is very difficult and inefficient to maintain.

## The Composite Approach

An effective remedy for these challenges is to partition the application into a number of discrete, loosely coupled, semi-independent components。Applications designed and built this way are often known as composite applications.

Composite applications provide many benefits, including the following:

They allow modules to be individually developed, tested, and deployed by different individuals or subteams;

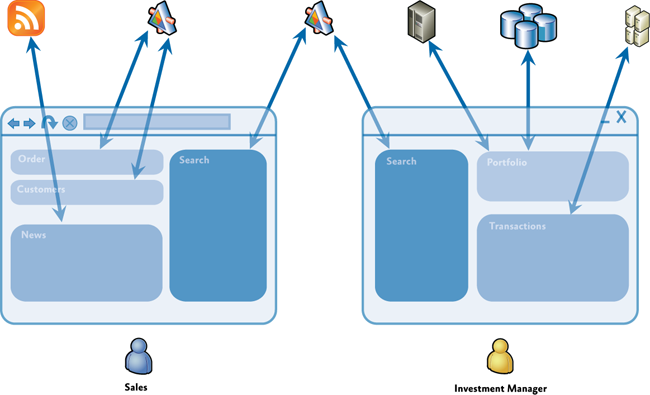
They provide a common shell composed of UI components contributed from various modules that interact in a loosely coupled way.

They promote reuse and a clean separation of concerns between the application's horizontal capabilities, such as logging and authentication, and the vertical capabilities, such as business functionality that is specific to your application.

it provides a cleaner separation between the UI and the business logic of the application

composite application is ideal for creating a rich end-user experience over disparate back-end systems

Composite application with multiple back-end systems



The Prism Library implements design patterns that embody important architectural design principles, such as separation of concerns and loose coupling.

## Prism Key Concepts

**Modules** Modules are packages of functionality that can be independently developed, tested, and (optionally) deployed.

Modules can be used to represent specific business-related functionality (for example, profile management) and encapsulate all the views, services, and data models required to implement that functionality. Modules can also be used to encapsulate common application infrastructure or services (for example, logging and exception management services) that can be reused across multiple applications

**Module catalog**. In a composite application, modules must be discovered and loaded at run time by the host application. In Prism, a module catalog is used to specify which modules to are to be loaded, when they are loaded, and in what order. The module catalog is used by the **ModuleManager** and **ModuleLoader** components, which are responsible for downloading the modules if they are remote, loading the module's assemblies into the application domain, and for initializing the module. Prism allows the module catalog to be specified in different ways, including programmatically using code, declaratively using XAML, or using a configuration file. You can also implement a custom module catalog if you need to.

**Shell**. The shell is the host application into which modules are loaded，it is typically unaware of the exact modules that it will host. It usually implements common application services and infrastructure, The shell also provides the top-level window or visual element that will then host the different UI components provided by the loaded modules.

**Views.**Views are used in conjunction with the MVVM pattern, which is used to provide a clean separation of concerns between the UI and the application's presentation logic and data. Views are used to encapsulate the UI and define user interaction behavior, Views use data binding to interact with view model classes.

**View models**. View models are classes that encapsulate the application's presentation logic and state. They are part of the MVVM pattern. View models encapsulate much of the application's functionality.. View models define properties, commands, and events, to which controls in the view can data-bind.

**Models**. Model classes encapsulate the application data and business logic. Models encapsulate data and any associated validation and business rules to ensure data consistency and integrity.

**Commands**. Commands are used to encapsulate application functionality in a way that allows them to be defined and tested independently of the application's UI. Prism provides the **DelegateCommand** class and the **CompositeCommand** class. The latter is used to represent a collection of commands which are all invoked together.

**Regions**. Regions are logical placeholders defined within the application's UI (in the shell or within views) into which views are displayed. Regions allow the layout of the application's UI to be updated without requiring changes to the application logic

**Navigation** Prism supports two styles of navigation: state-based navigation, where the state of an existing view is updated to implement simple navigation scenarios, and view-switching navigation, where new views are created and old views replaced within the application's UI.

**EventAggregator**. Components in a composite application often need to communicate with other components and services in the application in a loosely coupled way。**EventAggregator** component, which implements a pub-sub event mechanism, thereby allowing components to publish events and other components to subscribe to those events without either of them requiring a reference to the other.

**Dependency injection container**. Dependency injection allows component dependencies to be fulfilled at run time, and it supports extensibility and testability

**Services**. Services are components that encapsulate non-UI related functionality, such as logging, exception management, and data access. Services can be defined by the application or within a module.

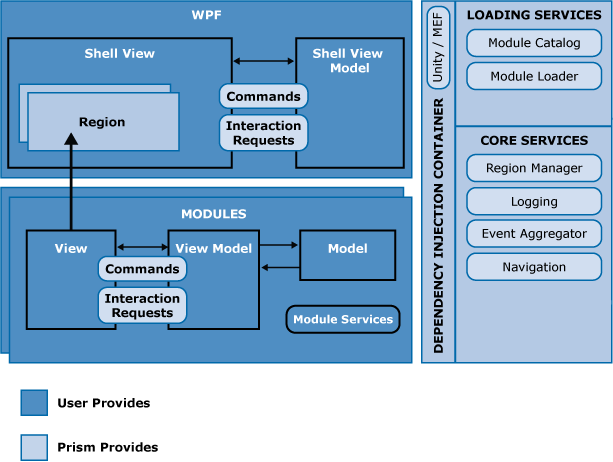
**Controllers**. Controllers are classes that are used to coordinate the construction and initialization of views that are to be displayed in a region within the application's UI

The controller will use Prism's view-switching navigation mechanism,

**Bootstrapper**. The **Bootstrapper** component is used by the application to initialize the various Prism components and services. It is used to initialize the dependency injection container to register any application-level components and services with it. It is also used to configure and initialize the module catalog and the shell's view and view model or presenter.

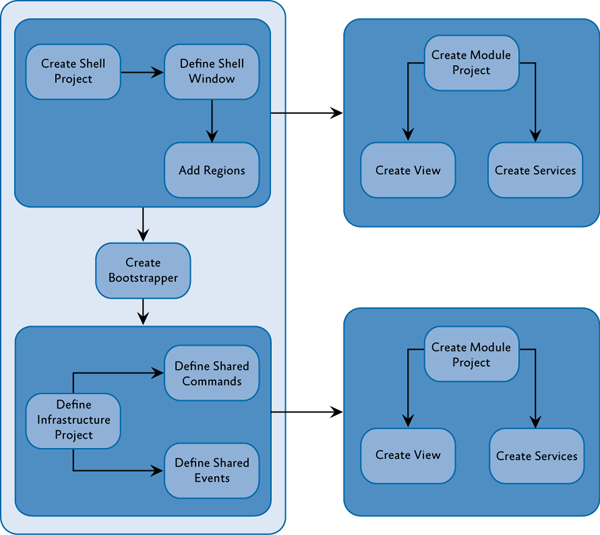
Prism is designed so that you can use any of the preceding capabilities and design patterns individually, or all together, depending on your requirements and your application scenario

Typical composite application architecture with the Prism Library



Most Prism applications consist of a shell application that defines regions for displaying top-level views and shared services that can be accessed by the loaded modules，The shell defines a suitable catalog to specify which modules are to be loaded at startup time , as appropriate. A dependency injection container is also defined

A Prism application typically consists of a shell project and multiple module projects.。the steps required to create a basic Prism application that consists of a single module that defines a single view are described.



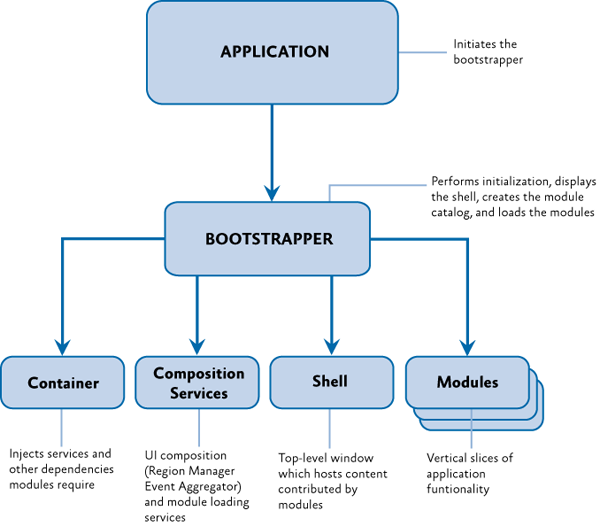
### Define the Shell

The application shell provides the basic layout for the application. This layout is defined using regions that modules can use to place views. Views, like shells, can use regions to define discoverable areas that content can be added to

### Create the Bootstrapper

The bootstrapper is the glue that connects the application with the Prism Library services and the Unity or MEF containers.

By default, the bootstrapper logs events using the .NET Framework **Trace** class. Applications can supply their logging service in their bootstrapper.



### Create the Module

The module contains the views and services specific to a piece of the application's functionality. Frequently, these are contained in separate assemblies and developed by separate teams

A module is denoted by a class that implements the **IModule** interface. These modules, during initialization, register their views and services and may add one or more views to the shell.

### Add a Module View to the Shell

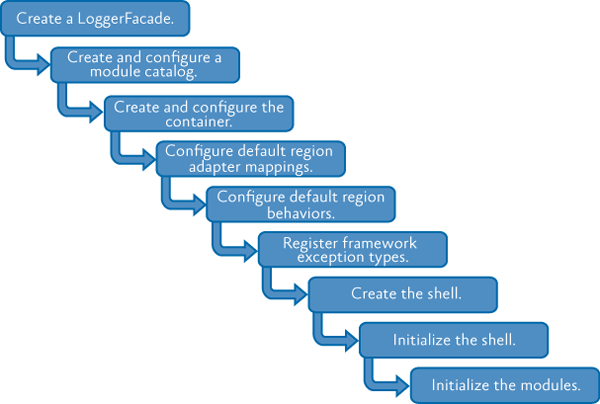
Modules take advantage of the shell's regions for placing content. During initialization, modules use the **RegionManager** to locate regions in the shell and add one or more views to those regions or register one or more view types to be created within those regions.

# 2: Initializing Applications Using the Prism Library 5.0 for WPF

A bootstrapper is a class that is responsible for the initialization of an application built using the Prism Library

The Prism Library includes a default abstract **Bootstrapper** base class that can be specialized for use with any container. Many of the methods on the bootstrapper classes are virtual methods

Basic stages of the bootstrapping process



The Prism Library provides some additional base classes, derived from **Bootstrapper**, that have default implementations that are appropriate for most applications. The only stages left for your application bootstrapper to implement are creating and initializing the shell.

## Dependency Injection

The Prism Library includes the **UnityBootstrapper** and **MefBootstrapper** classes, which implement most of the functionality necessary to use either Unity or MEF as the dependency injection container in your application

## Creating the Shell

It is the bootstrapper's responsibility to create the shell or the main window. This is because the shell relies on services, such as the Region Manager, that need to be registered before the shell can be displayed.

# Key Decisions

You will need to decide whether you are using MEF, Unity, or another container for your dependency injection container. This will determine which provided bootstrapper class you should use

You should think about the application-specific services you want in your application. These will need to be registered with the container.

Determine whether the built-in logging service is adequate for your needs

Determine how modules will be discovered by the application: via explicit code declarations, code attributes on the modules discovered via directory scanning, configuration, or XAML.

## Creating a Bootstrapper for Your Application

### Implementing the CreateShell Method

The **CreateShell** method allows a developer to specify the top-level window for a Prism application. The shell is usually the **MainWindow** or **MainPage**. Implement this method by returning an instance of your application's shell class. In a Prism application, you can create the shell object, or resolve it from the container, depending on your application's requirements.

### Implementing the InitializeShell Method

you may override the **InitializeShell** method for shell specific initialization.

For WPF applications, you will create the shell application object and set it as the application's main window, as shown here

protected override void InitializeShell()

{

Application.Current.MainWindow = Shell;

Application.Current.MainWindow.Show();

}

## Creating and Configuring the Module Catalog

Prism uses a concrete **IModuleCatalog** instance to keep track of what modules are available to the application, which modules may need to be downloaded, and where the modules reside.

The **Bootstrapper** provides a protected **ModuleCatalog** property to reference the catalog as well as a base implementation of the virtual **CreateModuleCatalog** method. this method can be overridden to provide a different **IModuleCatalog** instance instead,

In both the **UnityBootstrapper** and **MefBootstrapper** classes, the **Run** method calls the **CreateModuleCatalog** method and then sets the class's **ModuleCatalog** property using the returned value.

## Creating and Configuring the Container

# 3: Managing Dependencies Between Components Using the Prism Library 5.0 for WPF

To tie together these various pieces, applications based on the Prism Library rely on a dependency injection container

During the objects creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves their dependencies first

In some cases, the container itself is resolved as a dependency. For example, when using the Unity Application Block (Unity) as the container, modules have the container injected, so they can register their views and services with that container.

## Considerations for Using the Container

Consider whether it is appropriate to register and resolve components using the container: Consider whether the performance impact of registering with the container and resolving instances from it is acceptable in your scenario。the container's use of reflection for creating each entity.

 If there are many or deep dependencies, the cost of creation can increase significantly.

If the component does not have any dependencies or is not a dependency for other types, it may not make sense to put it in the container.

If the component has a single set of dependencies that are integral to the type and will never change, it may not make sense to put it in the container.？

Consider whether a component's lifetime should be registered as a singleton or instance:

If the component is a global service that acts as a resource manager for a single resource, such as a logging service, you may want to register it as a singleton.

 If the component provides shared state to multiple consumers, you may want to register it as a singleton.

Consider whether you want to configure the container through code or configuration:

* If you want to centrally manage all the different services, configure the container through configuration.
* If you want to conditionally register specific services, configure the container through code.
* If you have module-level services, consider configuring the container through code so that those services are registered only if the module is loaded.

## Registering

There are primarily two means for registering types and objects: through code or through configuration.

## Resolving

In general, when a type is resolved, one of three things happens:

If the type has not been registered, the container throws an exception.

If the type has been registered as a singleton, the container returns the singleton instance. If this is the first time the type was called for, the container creates it and holds on to it for future calls.

If the type has not been registered as a singleton, the container returns a new instance.