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;

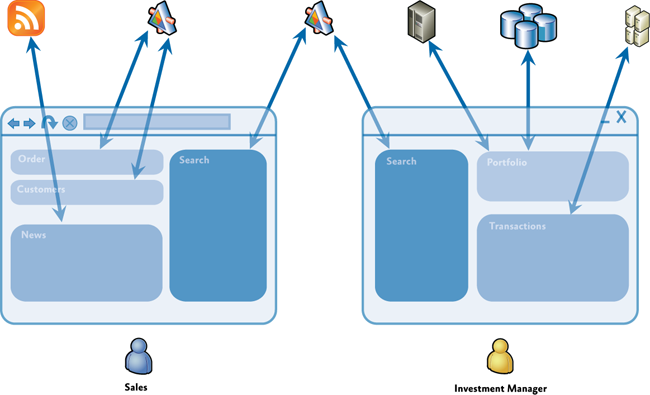
They provide a common shell composed of UI components contributed from various modules that interact in a loosely coupled way.

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.

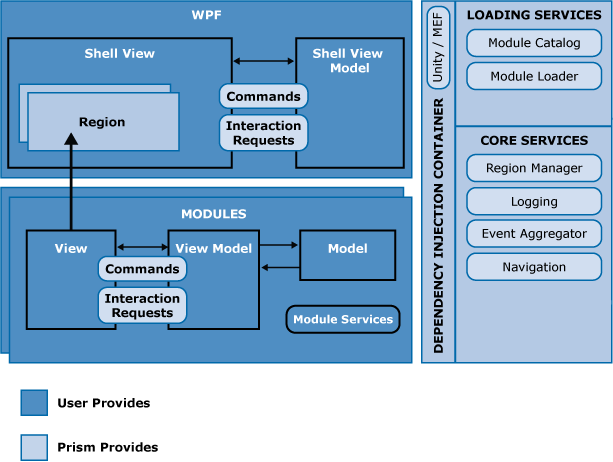
**Controllers**. Controllers are classes that are used to coordinate the construction and initialization of views that are to be displayed in a region within the application's UI

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.

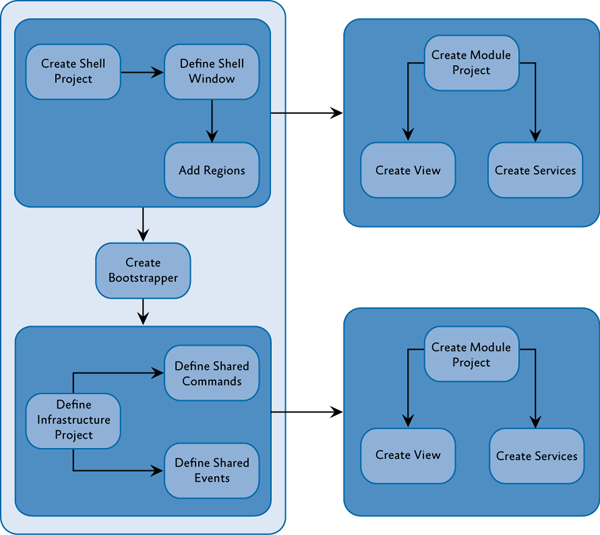
Prism is designed so that you can use any of the preceding capabilities and design patterns individually, or all together, depending on your requirements and your application scenario

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.



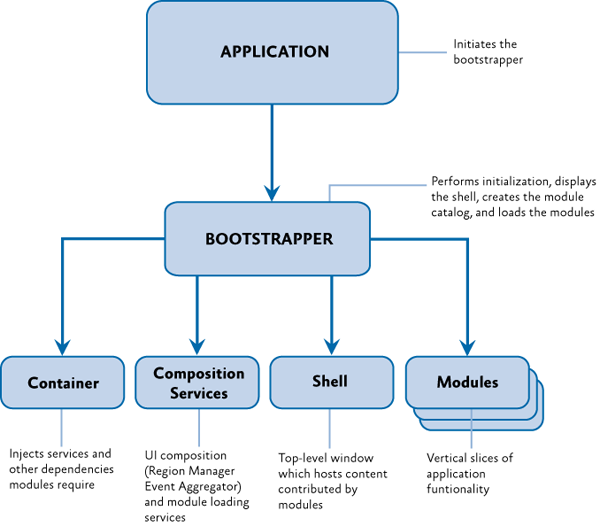
### Define the Shell

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.