# **Composite Application Guidance for WPF**

<http://msdn.microsoft.com/en-us/library/cc707819.aspx>

# Design Concepts

## **UI Composition**

There are many different challenges for composing the UI.

* Layout
* Commanding
* Eventing

### View Injection

the application contains a registry of res erved locations in the UI. A module can access one of the locations in the registry and use it to inject views. The view being injected does not have any specific knowledge of how it will be displayed in that location

### View Discovery

With View Discovery, the modules register their views (or presenters) in a central location, such as in the container, using a well-known interface. A shell service or a composite view then queries the container to discover the views that were registered. After they are discovered, the service lays out those views on the screen as appropriate, such as adding them to a panel or an items control. If after the application is loaded, additional views need to be displayed, such as a new order screen, the shell service or composite view should be notified to handle the display

### Commanding

In a composite WPF application, separated presentation patterns, such as **Model-View-Presenter**, **PresentationModel**, and **Model-View-ViewModel**, are used for decoupling the view from the business logic.

*By default CanExecute is requeried on focus change for RoutedCommand?*

In the RoutedCommand class, CanExecuteChanged is fired based on changes of state or focus in the UI.

#### Delegation

In this method, a command is used that delegates off its handling logic, either through events or delegates where it can be handled externally by a class such as a presenter, service, controller, and so on

#### Composition

Composition is a variation of delegation. In this approach, a composite command delegates off its handling logic to a set of child commands, such as in a **Save All** scenario described earlier.

### Eventing

In a composite application, components, such as presenters, services, and controllers, residing in different modules often need to communicate with one another based on state changes. This is a challenge due to the highly decoupled nature of a composite application because the publisher has no connection to the subscriber. Additionally, there may be threading issues because the publisher is on a different thread than the subscriber.The [Pub/Sub pattern](http://msdn.microsoft.com/en-us/library/ms978603.aspx) addresses these challenges

#### Event Services

In this method, an application-specific service raises standard .NET Framework events (works liek **EventMediator**). To add new events, the service and service interface need to be modified. This service is registered in the container where it can be accessed by the different modules in the system. The publisher and the subscriber reference the service interface and do not depend on one another.

#### Event Aggregation

This approach uses a generic event aggregator service registered in the container that holds a repository of event objects. The event object itself uses delegates instead of events. The advantage of this is that these delegates can be created at the time of publishing and immediately released, which does not prevent the subscribers from being garbage collected. Each event object contains a collection of subscribers it will publish to. New events can be added to the system without modifying the service. The event object can also automatically handle marshaling to the correct thread.

## Modularity

*Modularity* is designing a system that is divided into a set of functional units (named modules) that can be composed into a larger application. **A module represents a set of related concerns. It can include components, such as views or business logic, and pieces of infrastructure, such as services for logging or authenticating users.** Modules are **independent of one another** but can communicate with each other in a loosely coupled fashion.

**Application services** integrate components within the different modules and handle the communication with the user. The user sees an integrated view that looks like a single application.

When you develop in a modularized fashion, you structure the application into separate modules that can be individually developed, tested, and deployed by different teams. Modules can enforce separation of concerns by vertically partitioning the system and keeping a clean separation between the UI and business functionality.

The following are specific guidelines for developing a modular system:

* Modules should be opaque to the rest of the system and initialized through a well-known interface.
* Modules should not directly reference one another or the application that loaded them.
* Modules should use services to communicate with the application or with other modules.
* Modules should not be responsible for managing their dependencies. These dependencies should be provided externally, for example, through dependency injection.
* Modules should not rely on static methods that can inhibit testability.
* Modules should support being added and removed from the system in a pluggable fashion.

## Container

**dependency injection** container. The container creates instances of components that have service dependencies. During the component's creation, the container injects any dependencies that the component has requested into it. If those dependencies have not yet been created, the container creates and injects them first.

There are several advantages of using a container:

* A container removes the need for a component to have to locate its dependencies or manage their lifetime.
* A container allows swapping the implementation of the dependencies without affecting the component.
* A container facilitates testability by allowing dependencies to be mocked.
* A container increases maintainability by allowing new services to be easily added to the system.

public PositionModule(IUnityContainer container, IRegionManager regionManager)

{

\_container = container;

\_regionManagerService = regionManager;

}

public void Initialize()

{

RegisterViewsAndServices();

...

}

protected void RegisterViewsAndServices()

{

\_container.RegisterType<IAccountPositionService, AccountPositionService>();

\_container.RegisterType<IPositionSummaryView, PositionSummaryView>();

\_container.RegisterType<IPositionSummaryPresenter, PositionSummaryPresenter>();

…

}

Containers are used for two primary purposes, **namely registering** and **resolving**

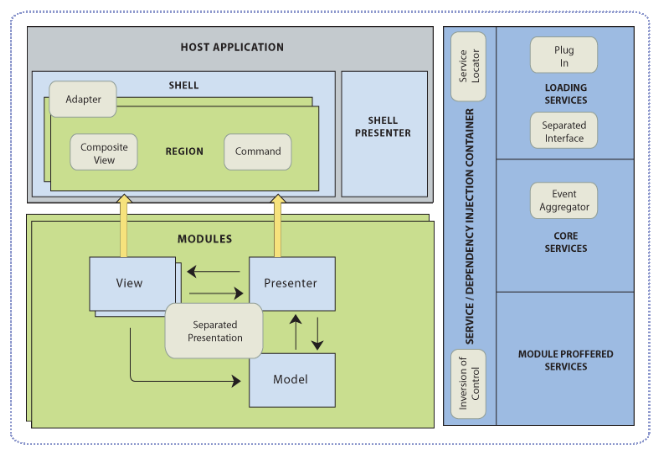
For services to be available to be injected, they need to be registered with the

container. Registering a service involves passing the container a service

interface and a concrete type that implements that service

After a service is registered, it can be resolved or injected as a dependency. When a service is being resolved, and the container needs to create a new instance, it will inject the dependencies into these instances.

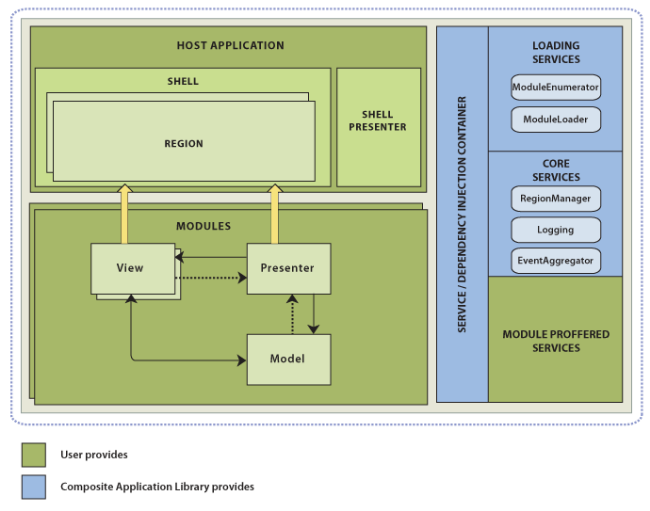
## Patterns in the Composite Application Library



*Composite application patterns*

* Composite User Interface patterns:
  + Composite and Composite View
  + Command
  + Adapter
* Modularity patterns:
  + Separated Interface and Plug In
  + Service Locator
  + [Event Aggregator](http://www.martinfowler.com/eaaDev/EventAggregator.html)
  + Façade
* Testability patterns:
  + Inversion of Control
  + Separated Presentation
    - Separated Presentation patterns are a category of patterns that focus on keeping the logic for the presentation separate from the visual representation. Primarily, this is done to allow testing of your presentation logic without the need to involve a visual representation. There are a number of specific implementations of these patterns, such as Model-View-Controller (MVC), Model-View-Presenter (MVP) variants, and Model-View-ViewModel (MVVM). For more information about MVC and MVP variants, see [GUI Architectures](http://martinfowler.com/eaaDev/uiArchs.html) on Martin Fowler's Web site. For more information about MVVM, see [Tales from the Smart Client](http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx) on MSDN. This guidance includes a description of the Supervising Controller and Presentation Model patterns. For more information, see [Supervising Controller](http://docs.google.com/prism2008june_e2e7ccc6-cb2c-4c22-bfee-b84e0ef3a01d.html) and [Presentation Model](http://docs.google.com/prism2008june_e1ffb712-316c-4211-ad70-52af67133717.html).  
      The Composite Application Library itself is intended to be neutral with respect to choice of Separated Presentation pattern. You can be successful with any of the patterns, although considering WPF's highly binding-oriented nature, patterns that support data-binding lend themselves better to WPF applications. The Stock Trader RI demonstrates the use of the Presentation Model pattern, an MVP variant that encapsulates the presentation logic and constructs a model to which the view can bind.

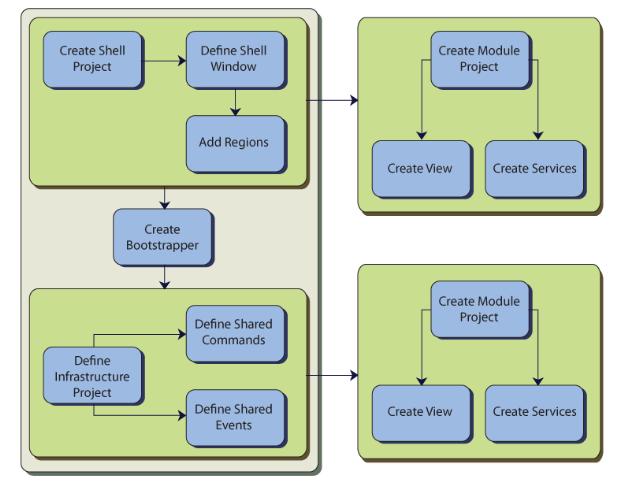
Composite Application Library Baseline Architecture



## The architectural pieces of a composite application are the following:

* **Shell**. This is the top level window to host different user interface components. The shell itself defines the layout structure, but it is typically unaware of the exact contents it will contain. It typically has minimal capability, so most of the application's functionality and content is provided by modules.
* **Shell presenter**. Any logic for the shell presentation is handled by the shell presenter. This follows the separated presentation pattern and helps separate the display of content from the user interface logic. This separation improves testability and maintainability.
* **Regions**. These are placeholders for content and host visual elements in the shell. These can be located by other components through the **RegionManager** to add content to those regions. Regions can also be used in module views to create discoverable content placeholders.
* **Modules**. These are separate sets of views and services, frequently logically related, that can be independently developed, tested, and optionally deployed. In many situations, these can be developed and maintained by separate teams. In a composite application, modules must be discovered and loaded, in the Composite Application Library this process is known as module enumeration and module loading.
  + **Module enumeration**. This is the process of locating individual modules for loading. This location can be done statically, through a configuration file, or by examining a directory. This includes the following types of enumeration:
    - **Static enumerators**. These allow the shell application to specify the set of modules in-code and are statically referenced from the shell application.
    - **Configuration-baseenumeration**. This specifies the modules to load in a configuration file.
    - **Directory sweep enumeration**. This examines a folder for the modules to load.
  + **Module loading**. This feature allows a component to specify, in code, when a module should be loaded. Modules discovered during enumeration may be immediately loaded or loaded on-demand.
  + **Initialization**. The **ModuleLoader** initializes each module when it is first loaded.
* **Views**. Views are responsible for displaying content on the screen. In a composite application, views are frequently the element of composition by the shell or other views. The following are considerations for use when developing views in WPF:
  + The user interface should be testable and implement appropriate design patterns for separations of concerns. Typically, this involves some type of binding-oriented design pattern, such as Supervising Controller or Presentation Model.
  + Windows Presentation Foundation (WPF) data binding should be used wherever possible.
  + WPF commands should be used for binding user interface actions.
* **Communication**. Different components in the application may need to communicate with one another whether they reside in the same or different modules. This needs to happen without the modules requiring hard dependencies on one another. The Composite Application Library provides mechanisms to do this with the **CompositeCommand** and **EventAggregator**. Communication strategies may need to consider thread-safety issues. The following describes the **CompositeCommand** and **EventAggregator**:
  + **CompositeCommand**. Frequently, when compositing views, commands those views support must also be composited. The composite command is a strategy to combine the execution of commands. This allows the command invoker to interact with a single command that affects multiple commands.
  + **EventAggregator**. In views that need to send an event to other views or components and do not require a response, use the **EventAggregator**. Multiple components can publish an event, and multiple subscribers can receive the event.
* **Services**. The application and modules expose services for their own and shared use. These are exposed through a service container that locates and, often, constructs the services. By default, the Composite Application Library uses the Unity container for this service location.

### A New Application Based on the Composite Application Library

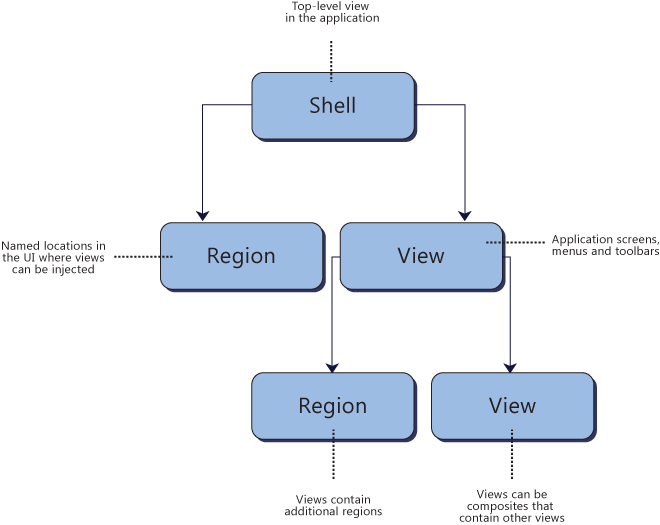


Activities for creating a WPF composite application

The core activities needed when starting a new composite WPF application are

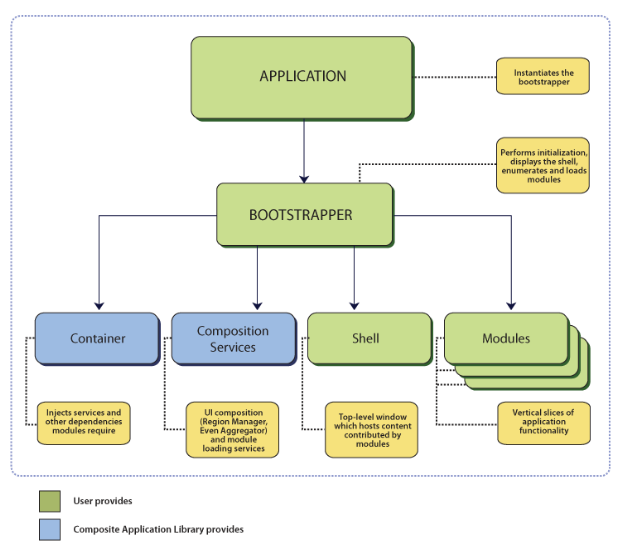
* Define the shell
* Create the bootstrapper
* Create a module
* Add a module view to the shell

### Define the shell

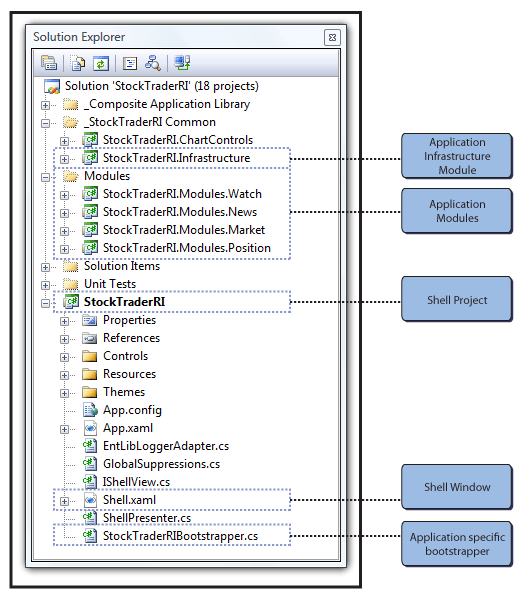


*Shells, views, and regions*

### Create the Bootstrapper



### Solution Organization



## Development Activities

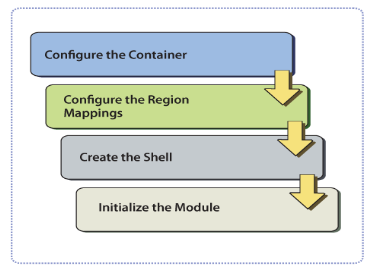


*Mapping of composite application concepts to How-to topics*

## Technical Concepts

### [**Bootstrapper**](http://msdn.microsoft.com/en-us/library/cc707847.aspx)

The bootstrapper is responsible for the initialization of an application built using the Composite Application Library. Having a bootstrapper gives you more control of how the Composite Application Library components are wired up to your application. CAL includes a default abstract **UnityBootstrapper** class that handles this initialization



*boot strapper process*

The bootstrapper is responsible for the initialization of an application built. The CAL includes a default abstract **UnityBootstrapper** class that handles this initialization using the Unity container.

**Configuring the Container**

[Containers](http://msdn.microsoft.com/en-us/library/dd458893.aspx) play a key role in an application created with the Composite Application Library. Both the Composite Application Library and the applications built on top of it depend on a container for injecting required dependencies

**Configuring the RegionAdapter Mappings**

During this phase, the default region adapter mappings are registered. These mappings are used by the region manager to associate the correct adapters for XAML-defined regions.

**Create the Shell**

**Initialize the Module**

### [**Container and Services**](http://msdn.microsoft.com/en-us/library/cc707875.aspx)

Dependency injection containers, often referred to as just "containers," are used to satisfy dependencies between components; satisfying these dependencies typically involves registration and resolution. The Composite Application Library provides support for the Unity Application Block (Unity) container, but it is not container-specific. Because the library accesses the container through the **IServiceLocator** interface, the container can be replaced.

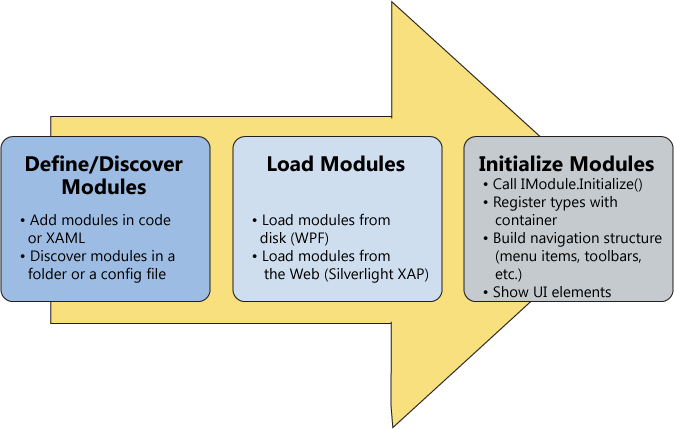
## Core Services

The following table lists the core non-application specific services in the Composite Application Library.

|  |  |
| --- | --- |
| Service interface | Description |
| **IModuleManager** | Defines the interface for the service that will retrieve and initialize the application's modules. |
| **IModuleCatalog** | Contains the metadata about the modules in the application. The Composite Application Library provides several different catalogs. For more information, see the Module technical concept. |
| **IModuleInitializer** | Initializes the modules. |
| **IRegionManager** | Registers and retrieves regions, which are visual containers for layout. |
| **IEventAggregator** | A collection of events that is loosely coupled between the publisher and the subscriber. |
| **ILoggerFacade** | A wrapper for a logging mechanism. The Stock Trader RI uses the Enterprise Library Logging Application Block, so you can choose your own logging mechanism. |
| **IServiceLocator** | Allows the Composite Application Library to access the container. If you want to customize or extend the library, this may be useful. |

### [**Module**](http://msdn.microsoft.com/en-us/library/cc707880.aspx)

A module is a logical unit in your application. These modules are defined in such a way that they can be discovered and loaded by the application at run time. Because modules are self-contained, they promote separation of concerns in your application. Modules can communicate with other modules and access services in a loosely coupled fashion. They reduce the friction of maintaining, adding, and removing system functionality. Modules also aid in testing and deployment.

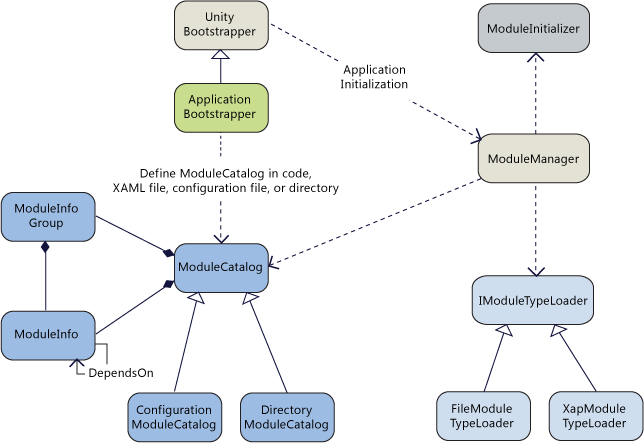


*Module loading process*

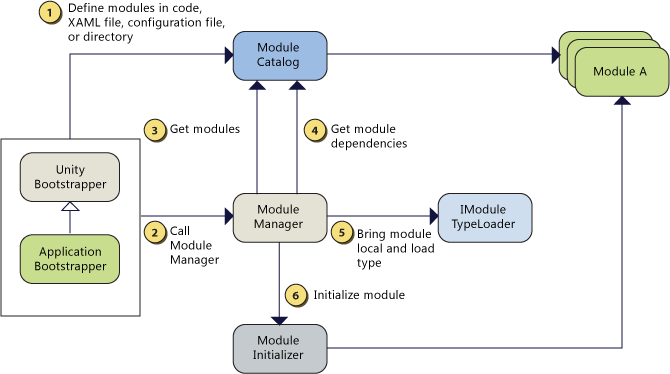
a possible module catalog may be used for loading the modules.

**When Initializing Modules**

* Register types with the container.
* Register view types with region names.
* Integrate the module with the application.
  + Add the module's views to the applications navigation structure
  + Subscribing to application level events or services



*Module Manager*



*Module Loading Process*

### [**UI Composition**](http://msdn.microsoft.com/en-us/library/dd458944.aspx)

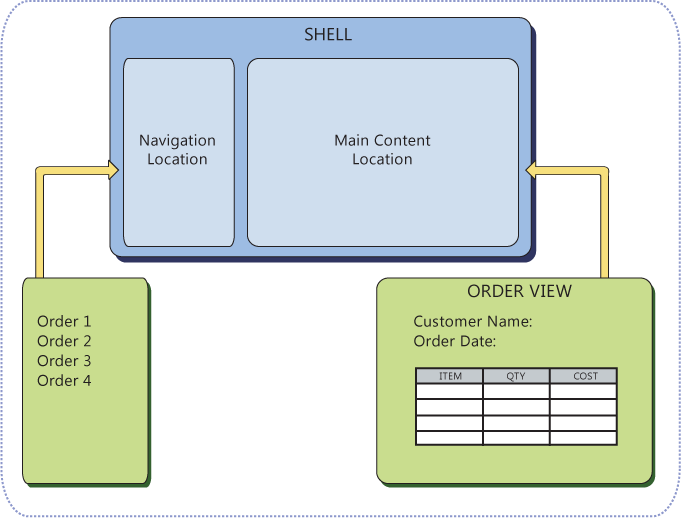
Views from multiple modules are displayed at run time in specific locations within the application's user interface (UI). The developer needs to define the locations where the views will appear and how the views will be created and displayed in those locations.

**Layout and Regions**

The developer defines where views will appear by defining a layout with named locations, known as regions, which act as placeholders within which one or more views will be displayed at run time. Modules can locate and add content to regions in the layout without exact knowledge of how and where the region is visually displayed. This allows the layout to change without affecting the modules that add the content to the layout.

Views can be created and displayed in regions either programmatically or automatically. The former is achieved through "view injection" and the latter through "view discovery."

The shell of the application defines the application's layout at the highest level, for example by specifying the locations for the main content and the navigation content, as illustrated in Figure 1. Layout within these high level views is similarly defined, allowing the overall UI to be recursively composed.



*A template shell*

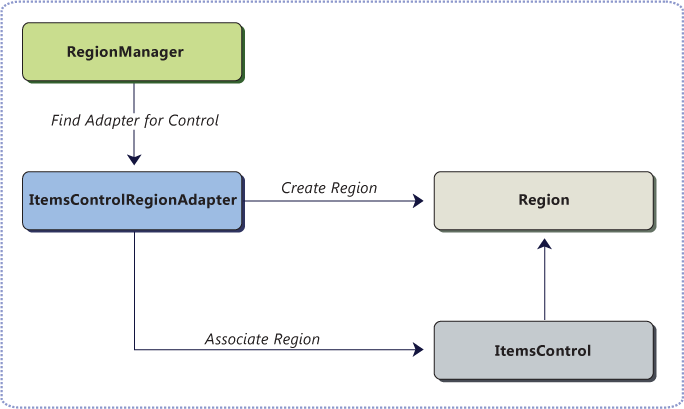
**View Discovery and View Injection**

In view discovery, you set up a relationship in the **RegionViewRegistry** between a region's name and the type of a view. When a region is created, the region looks for all the **ViewTypes** associated with the region and automatically instantiates and loads the corresponding views. Therefore, with view discovery, you do not have explicit control over when the regions' corresponding views are loaded and displayed.

In view injection, your code obtains a reference to a region and programmatically adds a view into it. Typically, this is done when a module initializes or as a result of a user action. Your code will query a **RegionManager** for a specific region by name and then inject views into it. With view injection, you have more control over when views are loaded and displayed; you also have the ability to remove views from the region

**Working with Regions**

Regions are enabled in the Composite Application Library through a region manager, regions, and region adapters.



*Region, control, adapter relationship*

**RegionContext** is useful when you want to share context between a parent view and child views that are hosted in a region. **RegionContext** is an attached property. You set the value of the context on the region control so that it can be made available to all child views that are displayed in that region control.

**Region Behaviors**

A region behavior is a class that attaches itself to a region to give the region some kind of functionality. This behavior is attached to the region and remains alive for as long as the region lives. It is easy to add custom region behaviors or replace existing behaviors, either on a system wide or a per region basis

### [**Shell and View**](http://msdn.microsoft.com/en-us/library/cc707902.aspx)

The shell is the main window of the application where the primary user interface (UI) content is contained.The shell may contain named [regions](http://msdn.microsoft.com/en-us/library/dd458944.aspx) where [modules](http://msdn.microsoft.com/en-us/library/dd458911.aspx) can specify the views that will appear. a view is just a collection of user interface elements that define part of the rendering of the user interface. It is a unit of encapsulation for defining the separable portions of your UI.

You do not have to have a distinct shell as part of your application architecture to use the Composite Application Library. For an existing application, you can alter your existing window definitions or controls to add regions or pull in views as needed.

The only thing included in the code-behind file is the implementation of the **IShellView** interface defined in the Shell project, as shown here.

 public interface IShellView

{

   void ShowView(); //Will be called by bootstrapper.

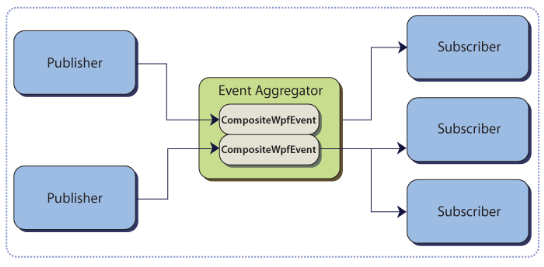
}

 You can define a view as a user control, data template, or even a custom control. A view encapsulates a portion of your user interface that you would like to keep as decoupled as possible from other parts of the application

a view that is supporting a specific set of functionality can get complicated. In that case, it may make sense to break up the view into several child views and have the parent view handle constructing itself using the child views as parts.

### [**Event Aggregator**](http://msdn.microsoft.com/en-us/library/cc707867.aspx)

, **EventAggregator** allows subscribers or publishers to locate a specific **EventBase**. The event aggregator also allows for multiple publishers and multiple subscribers.



The event aggregator is responsible for locating or building events and for keeping a collection of the events in the system.

public interface IEventAggregator

{

TEventType GetEvent<TEventType>() where TEventType : EventBase;

}

The **EventAggregator** will construct the event on its first access if it has not already been constructed.

The real work of connecting publishers and subscribers is done by the **CompositeWpfEvent** class. This is the only implementation of the **EventBase** class that comes out of the box in the Composite Application Library. This class maintains the list of subscribers and handles event dispatching to the subscribers.

The **CompositeWpfEvent** class is a generic class that requires the payload type to be defined as the generic type. This helps enforce, at compile time, that publishers and subscribers provide the correct methods for successful event connection.

The **CompositeWpfEvent** is intended to be the base class for an application's or module's specific events

public class TickerSymbolSelectedEvent : CompositeWpfEvent<string>{}

Subscribers can enlist with an event using one of the **CompositeWpfEvent** available **Subscribe** method overloads.

eventAggregator.GetEvent<TickerSymbolSelectedEvent>().Subscribe(ShowNews); (The ShowNews is the method name)

When subscribes, subscribes can decide the thread option (where the method should run)

* **Publisher**. Receive the event on the publishers' thread. This is the default setting.
* **Background**. Receive the event on a .NET Framework thread-pool thread.
* **UIThread**. Receive the event on the user interface thread.

Subscribers may not need to handle every instance of a published event. In these cases, the subscriber can subscribe and supply a delegate that filters the event before the registered handler is called. Frequently, this filter is supplied as a **lambda expression**, (***This is a great idea!***)

By default,**CompositeWpfEvent** maintains a **weak delegate(*!!!!*)** reference to the subscriber's handler and filter on subscription. This means the reference that **CompositeWpfEvent** holds to the subscriber will not prevent garbage collection of the subscriber. Yet we can still use Strong reference if for better performance as weak performance is slow.

Publishers raise an event by retrieving the event from the **EventAggregator** and calling the **Publish** method

EventAggregator.GetEvent<TickerSymbolSelectedEvent>().Publish(e.Value);

you can unsubscribe using your subscriber's handler directly or you can unsubscribe by using a subscription token.

compositeWpfEvent.Unsubscribe(FundAddedEventHandler);

Or

fundAddedEvent.Unsubscribe(subscriptionToken); (the token is returned during subscribtion).

### [**Commands**](http://msdn.microsoft.com/en-us/library/cc707894.aspx)

The commands in the Composite Application Library include **DelegateCommand** and **CompositeCommand**. These commands are different with the WPF's command, yet they implements the **ICommand** interface of WPF.

The **DelegateCommand** allows delegating the commanding logic instead of requiring a handler in the code behind. It uses a delegate as the method of invoking a target handling method.

The **DelegateCommand** uses its delegate to invoke a **CanExecute** method or **Execute** method on the target object when the command is invoked. Because the class is generic, it enforces compile-time checking on the command parameters

**DelegateCommand** accepts two constructor parameters, **executeMethod**and **canExecuteMethod**. Because the parameter types are generic delegates, the handlers can be easily hooked into the underlying controllers or presenters.

public class DelegateCommand<T> : ICommand

{

public DelegateCommand(Action<T> executeMethod, Func<T, bool> canExecuteMethod)

{

…

this.executeMethod = executeMethod;

this.canExecuteMethod = canExecuteMethod;

…

}

…

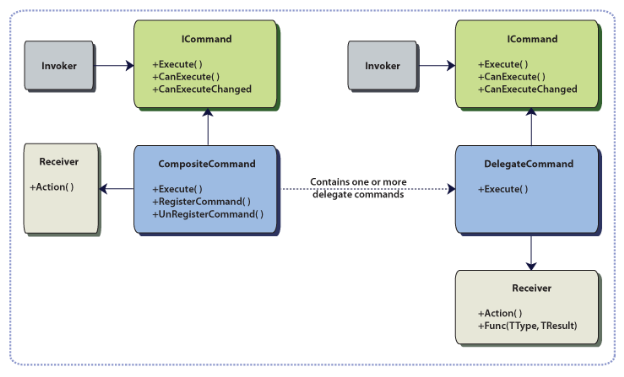
}

in some instances, you may want a delegate command to execute only if it is in a view that is currently active (selected)

To support this behavior, the **DelegateCommand** implements **IActiveAware**. This interface has an **IsActive** property, which can be set whenever the command becomes active. Whenever the property changes, the **DelegateCommand** raises the **IsActiveChanged** even

The **CompositeCommand** is a command that has multiple child commands.

When you call the **Execute** or the **CanExecute**method on the composite command, it calls the respective method on the child commands.



Composite commands are either registered or unregistered through the **RegisterCommand**and **UnregisterCommand**

***Why are WPF commands not used?***

WPF commands have a number of limitations. They are coupled to elements in the logical tree because they use routed events under the covers to deliver the command messages. This means you **cannot directly hook up a separate class, such as a presentation model, presenter, or controller, to be the direct command handler.** The view would have to be the routed command handler, and it would have to forward the call to the presenter or controller through a method call or event. Additionally, the **command handler that the routed event is delivered to is determined by the current focus in the UI.** This works fine if the command handler is at the window level, because the window is always in the focus tree of the currently focused element, so it gets called for command messages. However, it does not work for child views who have their own command handlers unless they have the focus at the time. Finally, **only one command handler is ever consulted with routed commands.** After one command handler is invoked (for **CanExecute** or **Execute**), no other handlers are consulted, even if they are in the focus tree of the focused element. For scenarios where multiple views (or their presenters) need to handle the command, there is no decoupled way to address it with routed commands.

***WPF Command only works on Visual Tree?***

***Can DelegateCommands be replaced with Routed Commands?***

No, because both are meant for two different purposes. Routed commands, such as **Cut**, **Copy**, and **Paste**, are meant for controls with command binding that live within the logical tree and that will have the focus for the intent of the command. They can also be used for general purposes if it is acceptable to put centralized command handling at the root window or page element and have it as part of the view. However, that approach does not scale for composite applications, so the **DelegateCommand** approach allows you to have the flexibility of multiple command handlers that live outside the logical tree.

***Can the order of execution of commands be set up inside the Composite commands?***

Currently, you cannot specify the order that commands are executed within **Composite** commands. Moreover, this is not required from a high level, because command handlers should be decoupled from one another and not rely on a specific invocation order. The workaround for this is the judicious usage of **DelegateCommands** and the implementation logic

[**Communication**](http://msdn.microsoft.com/en-us/library/cc707836.aspx)

When communicating between modules, you can use commanding, event aggregation, or shared services. Use the following to help decide which approach to use:

* Commanding. Use this in response to user gestures and custom enablement.
* Event aggregator. Use this to publish an event across modules.
* Shared services. Use this if neither of the preceding is applicable.

### [**Multi-Targeting**](http://msdn.microsoft.com/en-us/library/dd458864.aspx)

Means let the seperate the code and let some code can run on both WPF and sliverlight as they are not binary compatible.