Passaggio a Prism 7

Inizializzazione

La classe `App` deve derivare `PrismApplication`. Questa classe contiene sia il `Bootstrapper` (che non occorre né istanziare né avviare direttamente) che il `Container` delle varie istanze dell'applicazione. In particolare, il `Container` è di tipo `IContainerProvider` e consente di recuperare (e istanziare, se non è già stato fatto) tramite il metodo `Resolve<T>()` istanze di oggetti dichiarandone il tipo.

La classe `App` derivata deve implementare i metodi astratti:

- `Window CreateShell()`, che crea e mostra la main window dell'applicazione ritornando una sua istanza, ricavabile anche dal `Container`.
- `void RegisterTypes(IContainerRegistry containerRegistry)`, che consente di registrare istanze di oggetti passandole come parametri o dichiarandone il tipo. Gli oggetti possono essere registrati normalmente, registrati come istanze o registrati come singleton. Inoltre, è possibile registrare oggetti necessari per la navigazione (passaggio di dati tra view model diversi) e i dialoghi (popup).

Altri metodi utili da sovrascrivere (richiamando anche quelli di base) sono:

- `void OnStartup(StartupEventArgs e)`, per effettuare operazioni alla partenza dell'applicazione.
- `void OnExit(ExitEventArgs e)` , per effettuare operazioni alla chiusura dell'applicazione.
- `void ConfigureModuleCatalog(IModuleCatalog moduleCatalog)`, per aggiungere i vari moduli all'applicazione.
- `void ConfigureViewModelLocator()`, per associare ad ogni view model la sua view (utile per sfruttare la proprietà di auto localizzazione delle view ed evitare un fallimento nella convenzione `NomeView-NomeViewModel`).

IRegion

Le regioni consentono di associare ad una stessa porzione di UI diverse view. Questo avviene associando i vari tipi di oggetti view ad uno stesso nome di regione. Definendo nel codice xaml un elemento UI in questo modo:

```
<ContentControl prism:RegionManager.RegionName="NomeRegione"/>,
```

è possibile gestire il cambiamento di in modo automatico per tutte le regioni associate a "NomeRegione". Le regioni possono anche essere correlate ad un oggetto UI più specifico (e.g., `StackPanel`) aggiungendo una mappatura custom delle regioni nella sovrascrittura del metodo `void ConfigureRegionAdapterMappings(RegionAdapterMappings regionAdapterMappings)` nella classe `App`, dove l'oggetto da aggiungere deve derivare la classe `RegionAdapterBase<T>` (`T` è il tipo di oggetto UI) e sovrascrivere il metodo `Adapt(IRegion region, T regionTarget)` definendo come le regioni devono essere adattate all'oggetto UI.

Una view può essere associata in tre diversi modi ad una regione:

- Nella sovrascrittura del metodo `void OnInitialized(IContainerProvider containerProvider)` (vedi sezione IModule) si registra nell'oggetto `IRegionManager` (recuperabile dall'oggetto `IContainerProvider`) la view passandola per tipo tramite il metodo `void RegisterViewWithRegion(string name, Type type)`.
- Nel code behind della view dove è definita una regione, si dichiara nel costruttore che la view accetta un parametro di tipo `IRegionManager` e si registra la view allo stesso modo precedente ("view discovery").
- Nel code behind della view dove è definita una regione, si dichiara nel costruttore che la view accetta dei parametri di tipo `IContainerExtension` e `IRegionManager`, si estrae dal container un oggetto del tipo della view da mostrare nella regione e si aggiunge alla regione con il nome corrispondente, ricavandola dall'oggetto `IRegionManager` ("view injection").

Iscrivendosi all'evento `CollectionChanged` della proprietà `Regions` dell'oggetto `IRegionManager` si può monitorare il ciclo di vita delle regioni (eventi Add-Remove) e per ogni regione, iscrivendosi all'evento `CollectionChanged` della proprietà `Views`, si può monitorare il ciclo di vita delle view registrate nelle diverse regioni.

IModule

I diversi moduli devono avere una classe principale di tipo `IModule`, che richiede di implementare i metodi:

- `void OnInitialized(IContainerProvider containerProvider)`, dove è possibile, tra le altre cose, registrare nell'oggetto la view principale del modulo passandola per tipo tramite il metodo `void RegisterViewWithRegion(string name, Type type)`.
- `void RegisterTypes(IContainerRegistry containerRegistry)`, che consente di integrare lo stesso metodo contenuto nella classe `App` con oggetti specifici per il modulo.

I moduli possono essere dichiarati in diversi modi: nel metodo `void ConfigureModuleCatalog(IModuleCatalog moduleCatalog)` possono essere aggiunti uno ad uno per tipo o individuati dichiarando la cartella in cui sono contenuti oppure possono essere dichiarati nel file app.config individuando assembly, nome e modalità di inizializzazione (quella di default è "when available", in contrapposizione a "on demand").

ViewModelLocator

Il `ViewModelLocator` è uno strumento che consente di associare in maniera automatica una view al al suo view model. Per attivarlo, è necessario inserire nello xaml della view le righe:

```
xmlns:prism="http://prismlibrary.com/"
prism:ViewModelLocator.AutoWireViewModel="True"
```

La localizzazione avviene sulla base di una convenzione: l'oggetto della view (e.g., NomeView) si trova in un assembly che termina con .Views e l'oggetto del view model si chiama NomeViewModel e si trova nello stesso assembly ma terminante con .ViewModels. Questa convenzione può anche essere personalizzata sovrascrivendo il metodo `void ConfigureViewModelLocator()` nella classe `App` e passando al metodo `SetDefaultViewTypeToViewModelTypeResolver(Func<T,T> resolver)` della classe statica `ViewModelLocationProvider` un metodo che ritorna l'associazione tra il tipo della

view e il tipo del view model. Inoltre, è anche possibile bypassare qualsiasi convenzione e dichiarare direttamente l'associazione tra tipo di view e tipo di view model chiamando in modo equivalente i metodi della stessa classe:

- `Register(string viewTypeName, Type viewModelType)` ("type-type"),
- `Register(string viewTypeName, () => Container.Resolve<TViewModel>())` ("type-factory"),
- `Register<TView>(() => Container.Resolve<TViewModel>())` ("generic-factory"),
- `Register<TView,TViewModel>()` ("generic-generic").

ICommand

La classe `DelegateCommand` implementa l'interfaccia `ICommand` e può essere usata per associare elementi della view ad azioni del view model. Il suo costruttore richiede due parametri (il secondo facoltativo): un oggetto ("Execute") di tipo `Action` che rappresenta l'azione da svolgere quando avviene l'interazione con l'elemento della view e un oggetto ("CanExecute") di tipo `Func<bool>` che stabilisce se questa azione può essere eseguita. Per una maggiore sinteticità e pulizia del codice, questi oggetti possono essere istanziati includendo anche una chiamata ai metodi :

- `ObservesProperty(Expression<Func<bool>> propertyExpression)`, che permette di associare la chiamata al metodo "CanExecute" ad una proprietà che implementa `INotifyPropertyChanged` in modo che sia invocata quando il valore di questa cambia;
- `ObservesCanExecute(Expression<Func<bool>> propertyExpression)`, che, oltre ad associare la chiamata a "CanExecute" al cambiamento della proprietà, identifica anche il suo valore di ritorno con quello della proprietà stessa.

La classe `CompositeCommand` consente invece di registrare più comandi in unico oggetto per esporli in modo collettivo. Il tipico esempio è quello di un comando "save" generico che è bindato alla UI e a cui sono registrati i diversi comandi "save" associati alle varie view presenti, in modo che quando si invoca il `CompositeCommand` tutti i "save" vengano invocati. Se si vuole che venga invocato esclusivamente il comando associato alla view attiva, queste o i diversi view model devono implementare l'interfaccia `IActiveAware`. Anche la classe `DelegateCommand` implementa questa interfaccia e la classe `CompositeCommand` può essere inizializzata specificando `true` nel suo costruttore in modo che monitori l'attività di tutti comandi quando viene invocato uno dei suoi metodi. Un `CompositeCommand` è tipicamente reso disponibile globalmente a tutta l'applicazione inserendolo in una interfaccia generica, registrando un singleton di questa interfaccia nella classe `App` e dichiarando che i costruttori dei view model dove è necessario accettano come parametro un'istanza di questa interfaccia ("DependencyInjection")

IEventAggregator

L'interfaccia `IEventAggregator` fornisce un punto unico per tutta l'applicazione tramite il quale gli oggetti possono iscriversi a e pubblicare eventi globali. Gli oggetti che identificano questi eventi devono derivare dalla classe `PubSubEvents<T>`, dove `T` è il tipo dell'oggetto che identifica il parametro dell'evento. L'istanza di `IEventAggregator` è registrata in automatico dalla classe `PrismApplication` e può essere acquisita in una classe tramite "DependencyInjection". Successivamente, la classe può recuperare da essa il tipo di evento che occorre e iscriversi tramite il

metodo `Subscribe<TEvent>(Action<TEventArgs> action)` specificando la callback da eseguire oppure invocare un evento con il metodo `Publish(TEventArgs args)`.

INavigationAware

View associate alla stessa regione possono essere intercambiate chiamando dal view model principale il metodo `RequestNavigate(string regionName, string source)` dall'oggetto di tipo `IRegionManager`. I parametri da specificare riguardano il nome della regione a cui applicare la navigazione e il tipo della view registrata da mostrare. Le view devono essere registrate per la navigazione nel metodo `RegisterTypes(IContainerRegistry containerRegistry)` della classe che implementa `IModule` chiamando il metodo `RegisterForNavigation<TView>()` dal `containerRegistry`. I view model delle view registrate possono implementare (facoltativo) `INavigationAware` per compiere delle azioni in corrispondenza dell'attivazione di una view. I metodi che prevede l'interfaccia sono:

- `void OnNavigatedTo(NavigationContext navigationContext)`, per compiere azioni non appena la view corrispondente viene attivata.
- `void OnNavigatedFrom(NavigationContext navigationContext)`, per compiere azioni appena prima che la view corrispondente venga disattivata.
- `bool IsNavigationTarget(NavigationContext navigationContext)`, per definire se la view corrispondente può gestire la richiesta di navigazione. Questo metodo è utile per definire quando riutilizzare o navigare verso una view già esistente: se si ritorna sempre `true`, la view mostrata non mai viene ricreata.

Per passare parametri alla view verso cui si naviga, si può chiamare il metodo `RequestNavigate(string regionName, string source, NavigationParameters parameters)` dopo aver creato un oggetto di tipo `NavigationParameters`, che consente di aggiungere coppie chiave-valore, dove la chiave è di tipo `string`. Per recuperare il parametro, si può ricavare l'oggetto dal `NavigationContext` passato nei metodi di `INavigationAware` e richiedere l'oggetto associato alla chiave.

Per impostare la navigazione con richiesta di conferma si può implementare l'interfaccia `IConfirmNavigationRequest`, la quale estende `INavigationAware`. Ai metodi precedenti viene aggiunto il metodo:

- `void ConfirmNavigationRequest(NavigationContext navigationContext, Action<bool> continuationCallback)`, dove può essere specificato se proseguire oppure no chiamando la callback passandole come parametro `true` o `false` in base al risultato dell'interazione con l'utente.

Recuperando dalla proprietà `NavigationService` dell'oggetto di tipo `NavigationContext` l'oggetto `IRegionNavigationJournal`, è possibile poi ottenere un servizio che registra le azioni di navigazione e consente di navigare avanti (metodo `GoForward()`) e indietro (metodo `GoBack()`) tra le view appena mostrate.

IDialogService

L'oggetto di tipo `IDialogService` registrato all'avvio dell'applicazione occorre per mostrare popup sopra ad altre finestre dell'applicazione. Esiste un implementazione standard che può anche essere personalizzata. Questa interfaccia richiede i metodi:

- `void Show(string name, IDialogParameters parameters, Action<IDialogResult> callback)`, per mostrare senza disattivare la view padre una view registrata con il nome `name`, passandole i parametri `parameters` (l'implementazione di default `DialogParameters` di `IDialogParameters` estende `NavigationParameters`, perciò funziona allo stesso modo) e un'azione che accetta come parametro un oggetto `IDialogResult` per gestire la chiusura del dialogo in base al risultato dell'interazione utente.
- `void ShowDialog(string name, IDialogParameters parameters, Action<IDialogResult> callback)`, che funziona allo stesso modo ma disattivando la view padre.

In alternativa, è possibile creare una classe statica che espone dei metodi personalizzati che al loro interno chiamano questi metodi sull'oggetto `IDialogService` di default, che deve essere passato come parametro.

Per essere mostrata, una view deve prima essere registrata nel metodo `void RegisterTypes(IContainerRegistry containerRegistry)` della classe `App` attraverso il metodo `RegisterDialog<TView, TViewModel>(string name)`. In sostanza, questo rende disponibile la view all'oggetto `IDialogService`, che può recuperarla attraverso il nome passato come parametro. Inoltre, viene anche associato il view model corrispondente, il quale deve implementare l'interfaccia `IDialogAware`. Questa richiede di esporre:

- la proprietà `string Title`, per dichiarare il titolo del dialogo;
- l'evento `event Action<IDialogResult> RequestClose`, per chiudere il dialogo e invocare la callback passata come parametro impostando il risultato del dialogo;
- il metodo `void OnDialogOpened(IDialogParameters dialogParameters)`, per gestire l'apertura del dialogo e i parametri passati;
- il metodo `void OnDialogClosed()`, per gestire la chiusura del dialogo;
- il metodo `bool CanlCloseDialog()`, per stabilire se il dialogo può essere chiuso o meno.

Infine, è possibile anche sovrascrivere la finestra di default dove vengono collocate i dialoghi nel metodo `void RegisterTypes(IContainerRegistry containerRegistry)` della classe `App` sfruttando il metodo `RegisterDialogWindow<TWindow>()`, dove `TWindow` deve implementare l'interfaccia `IDialogWindow`, la quale richiede solamente di esporre una proprietà di tipo `IDialogResult`.