Model-View-ViewModel (MVVM)

Model-View-ViewModel (MVVM) è un modello per la progettazione di app che consente di separare il codice dell'interfaccia utente dal resto. Con MVVM puoi definire l'interfaccia utente in modo dichiarativo (ad esempio tramite XAML) e usare markup di associazione dati per collegarla ad altri livelli contenenti dati o comandi utente. L'infrastruttura di associazione dati fornisce un accoppiamento debole che permette di mantenere sincronizzati i dati collegati e l'interfaccia utente, oltre a indirizzare tutti gli input dell'utente ai comandi appropriati.

Il modello MVVM consente di organizzare il codice in modo da permettere la modifica di singole parti senza influire sulle altre. Questo offre numerosi vantaggi, ad esempio:

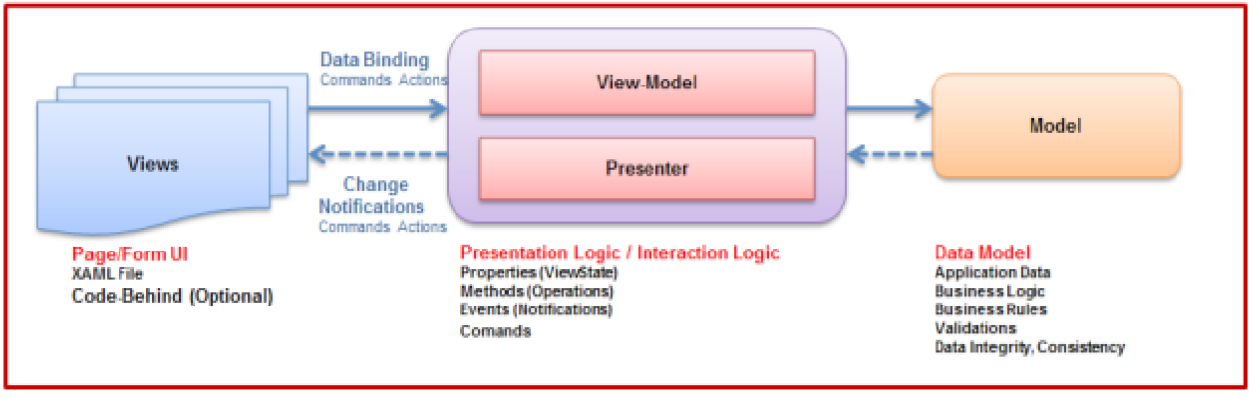
- Possibilità di adottare uno stile di codifica iterativo e basato sull'esplorazione.
- Semplificazione degli unit test.
- Possibilità di sfruttare più efficacemente strumenti di progettazione.
- Supporto della collaborazione in team.

Nelle app con una struttura più convenzionale, invece, l'associazione dati viene utilizzata solo per gli elenchi e i controlli di testo, e risponde all'input dell'utente gestendo gli eventi esposti dai controlli. I gestori di eventi sono strettamente accoppiati ai controlli e in genere contengono codice che manipola direttamente l'interfaccia utente. In questo caso è molto più difficile o impossibile sostituire un controllo senza aggiornare il codice dei gestori di eventi.

Livelli di un'app

Le app basate sul modello MVVM sono suddivise nei livelli seguenti:

- Il livello del **modello** che include tutto il codice della logica di base de definisce i tipi necessari per modellare il dominio. Questo livello è indipendente dalla visualizzazione e dal modello di visualizzazione.
- Il livello di **visualizzazione** definisce l'interfaccia utente tramite un markup dichiarativo. Il markup di associazione dati definisce la connessione fra componenti specifici dell'interfaccia utente e i vari membri del modello di visualizzazione, e a volte con i membri del modello stesso.
- Il livello del **modello di visualizzazione** fornisce le destinazioni dell'associazione dati per la visualizzazione. In molti casi il modello di visualizzazione espone direttamente il modello e fornisce membri che eseguono il wrapping di membri specifici del modello. Il modello di visualizzazione può anche definire membri che consentono di tenere traccia dei dati rilevanti per l'interfaccia utente ma non per il modello, ad esempio l'ordine di visualizzazione degli elementi di un elenco.



Vantaggi dei livelli

Grazie a questa separazione in livelli il codice risulta molto più semplice da capire, perché il codice relativo a funzionalità specifiche rimane distinto dal resto, risultando più facile da comprendere e da riutilizzare in altre app. Nel caso di un gioco, ad esempio, se si è interessati solo ai concetti relativi all'interfaccia utente e non all'intelligenza artificiale del gioco, si può ignorare il livello del modello.

Un altro vantaggio della separazione dell'UI è rappresentato dalla semplificazione degli unit test automatizzati.

In generale, un'architettura strettamente accoppiata complica notevolmente i cambiamenti e la diagnosi degli errori. Il principale vantaggio di un'architettura disaccoppiata consiste nella possibilità di isolare l'impatto delle modifiche. Pertanto, sperimentare nuove funzionalità, correggere bug e incorporare i contributi dei collaboratori risulta meno problematico.

Modello MVVM base e avanzato

Esistono numerosi framework e tecniche MVVM avanzate (PRISM, MVVM Light, Cinch, ...) che risultano utili con l'aumentare della complessità di un'app; per apprendere il modello di base è consigliabile seguire l'uso delle tecniche MVVM di base che contribuisce a mantenere la flessibilità del codice durante le fasi iniziali dello sviluppo, evitando di aggiungere ulteriori infrastrutture finché non diventano effettivamente necessarie.

Prerequisiti

Ci sono alcuni concetti importanti che si utilizzano intensamente, e si dovrebbe familiarizzare con loro:

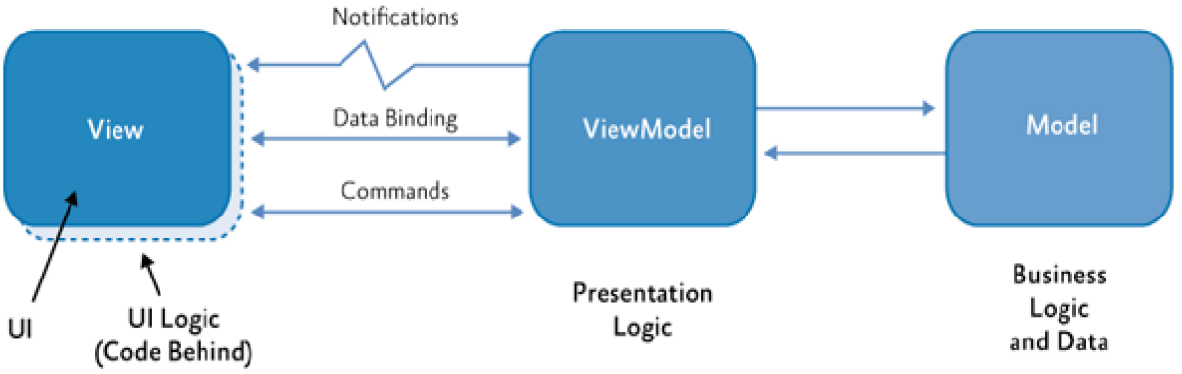
Concetto	Descrizione
XAML (Extensible Application Markup Language)	La lingua per definire in modo dichiarativo e inizializzare l'interfaccia utente in applicazioni WPF.
Associazione dati (Data binding)	Viene definito come gli elementi dell'interfaccia utente sono connessi ai componenti e dati in WPF.
Risorse (Resources)	Questi sono stili, modelli di dati e modelli di controllo che vengono creati e gestiti in WPF.
Comandi (Commands)	Definisce come gesti dell'utente e input sono collegati ai controlli.
Controlli utente (User controls)	Si tratta di componenti che forniscono un comportamento personalizzato o aspetto personalizzato.
Proprietà di dipendenza (Dependency properties)	Questi sono estensioni di proprietà del sistema runtime (CLR) per consentire l'impostazione di proprietà e il monitoraggio a sostegno di associazione dati, comandi instradati ed eventi.
Comportamenti (Behaviors)	I comportamenti sono oggetti che incapsulano funzionalità interattiva che può essere facilmente applicato a controlli dell'interfaccia utente.

Responsabilità di classe e caratteristiche

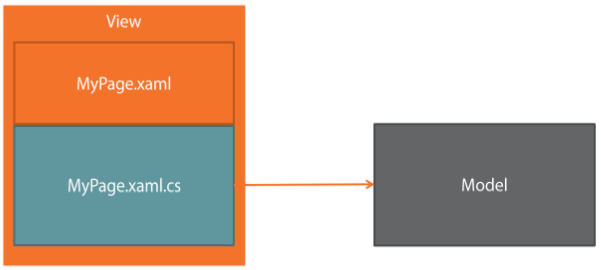
Il pattern MVVM è una variante del modello di presentazione del modello MVP, ottimizzato per sfruttare alcune delle funzionalità di WPF, come ad esempio l'associazione dati, modelli di dati, comandi, e comportamenti.

Nel modello MVVM, la View incapsula l'interfaccia utente e ogni logica dell'interfaccia utente, il ViewModel incapsula la logica di presentazione e lo stato, mentre il Model incapsula la logica di business e i dati. La View interagisce con il ViewModel attraverso associazione dati, i comandi e gli eventi di notifica di modifica. Il ViewModel

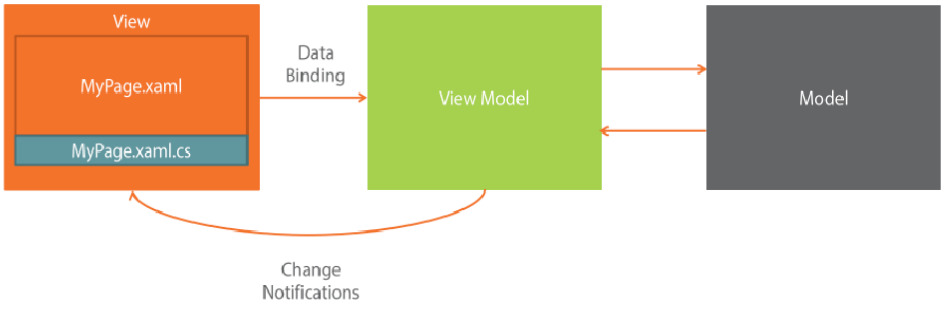
fa richieste, osserva, e coordina gli aggiornamenti al modello, convertendo, convalidando e aggregando i dati come necessario per la visualizzazione nella View.



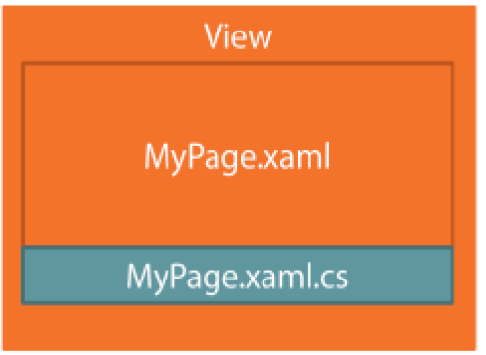
Senza MVVM



Con MVVM



La classe View



La responsabilità della View è quello di definire la struttura e l'aspetto di ciò che l'utente vede sullo schermo. Idealmente, il code-behind di una vista contiene solo un costruttore che chiama il metodo InitializeComponent. In alcuni casi, il code-behind può contenere codice per la logica UI che implementa il comportamento visivo che è difficile o inefficiente esprimere in Extensible Application Markup Language (XAML), come ad esempio animazioni complesse, o quando il codice ha bisogno di manipolare direttamente gli elementi visivi.

In WPF, le espressioni nella visualizzazione di associazione dati vengono valutati sulla base del data context. In MVVM, il contesto dati della View è impostato sul ViewModel. Il ViewModel implementa proprietà e comandi a cui la View può legarsi e far visualizzare eventuali cambiamenti di stato attraverso eventi di notifica di modifica. C'è in genere una relazione uno-a-uno tra una View e il suo ViewModel.

In genere, le View sono classi derivate da Control o UserControl. Tuttavia, in alcuni casi, la View può essere rappresentato da un modello di dati (data template), che specifica gli elementi dell'interfaccia utente da utilizzare per rappresentare visivamente un oggetto.

I data template possono essere pensati come View che non hanno alcun codice. Essi sono progettati per legarsi a uno specifico tipo di ViewModel ogni volta che si è richiesto la visualizzazione nell'interfaccia utente.

In WPF, è possibile associare un modello di dati con un ViewModel a livello di applicazione. WPF farà quindi applicare automaticamente il modello di dati al ViewModel ogni volta che vengono visualizzati nell'interfaccia utente. Il modello di dati può essere definito nel controllo che lo utilizza o in un dizionario risorse al di fuori della View e letto da un dizionario risorse della View.

Per riassumere, la vista ha le seguenti caratteristiche principali:

- La View è un elemento visivo, ad esempio una finestra, pagina, controllo utente, o modello di dati. La vista definisce i controlli contenuti nella View e il loro layout visivo e styling.
- La View fa riferimento al ViewModel attraverso la proprietà DataContext. I controlli nella visualizzazione dati sono legati alle proprietà e comandi esposti dal ViewModel. L'associazione può avvenire o nel costruttore, o nello XAML in **<UserControl.DataContext>**
- La View può personalizzare il comportamento dell'associazione dati tra la View e il ViewModel. Ad esempio, la View può utilizzare convertitori di valore per formattare i dati da visualizzare nell'interfaccia utente, o può utilizzare le regole di convalida per fornire la convalida dei dati di input aggiuntivo per l'utente (nella sezione **<UserControl.Resources>**).
- La View definisce e gestisce il comportamento visivo, come ad esempio animazioni o transizioni che possono essere attivati da un cambiamento di stato nel ViewModel o tramite l'interazione dell'utente con l'UI.
- Il codice della View può definire la logica dell'interfaccia utente per implementare comportamento visivo difficile da esprimere in XAML o che richiede riferimenti diretti ai controlli dell'UI specifici definiti nella View.

La classe ViewModel

Il ViewModel nel modello MVVM incapsula la logica di presentazione dei dati per la View. Non ha alcun riferimento diretto alla View o qualsiasi conoscenza dell'implementazione della View. Il ViewModel implementa proprietà e comandi a cui la View può associare dati e notificare la View di eventuali modifiche allo stato attraverso eventi di notifica cambiamento. Le proprietà e i comandi che il ViewModel fornisce, definiscono la funzionalità offerta dall'interfaccia utente, ma la View determina come tale funzionalità deve essere visualizzata.

Il ViewModel ha compito di coordinare l'interazione della View con tutte le classi del Model che sono necessarie. In genere, vi è una relazione uno-a-molti tra il ViewModel e le classi del Model. Il ViewModel può scegliere di esporre classi del Model direttamente alla View in modo che i controlli nella visualizzazione dati possono legarsi direttamente a loro.

Il ViewModel può convertire o manipolare i dati del Model in modo che possa essere facilmente consumato dalla View. Il ViewModel può definire proprietà aggiuntive per supportare specificamente la View; queste proprietà non sarebbero normalmente parte del (o non possono essere aggiunte al) Model. Ad esempio, il ViewModel può combinare il valore di due campi, o può calcolare il numero di caratteri rimanenti per i campi con una lunghezza massima. Il ViewModel può anche implementare la logica di convalida dati per garantire la coerenza dei dati.

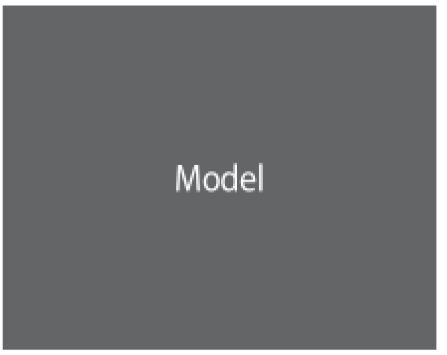
Il ViewModel può anche definire gli stati logici che la View può utilizzare per fornire cambiamenti visivi nell'interfaccia utente. La View può definire il layout o lo styling che riflettono lo stato del ViewModel. Ad esempio, il ViewModel può definire uno stato che indica che i dati sono presentati in modo asincrono a un servizio web. La View può visualizzare un'animazione durante questo stato per fornire un feedback visivo all'utente.

In genere, il ViewModel definirà comandi o azioni che possono essere rappresentati nell'interfaccia utente e che l'utente può richiamare. Un esempio comune è quando il ViewModel vista fornisce un comando Invia che permette all'utente di inviare dati a un servizio Web o di un repository di dati. La View può scegliere di rappresentare quel comando con un pulsante in modo che l'utente può fare clic sul pulsante per inviare i dati. In genere, quando il comando non è disponibile, la sua rappresentazione interfaccia utente associata viene disabilitata. Comandi forniscono un modo per incapsulare le azioni dell'utente e di separarli dalla loro rappresentazione visiva nell'UI.

Per riassumere, il ViewModel ha le seguenti caratteristiche principali:

- Il ViewModel è una classe non visiva e non deriva da alcuna classe di base WPF. Esso incapsula la logica di presentazione necessaria per supportare un caso d'uso o di task nell'applicazione. Il ViewModel è verificabile in modo indipendente della View e dal Model.
- Il ViewModel in genere non fa riferimento direttamente alla View. Implementa proprietà e comandi per i quali i dati binding con la View. Informa di eventuali cambiamenti di stato tramite le interfacce **INotifyPropertyChanged** e INotifyCollectionChanged.
- Il ViewModel coordina l'interazione della View con il Model. Può convertire o manipolare i dati in modo che possa essere facilmente consumato dalla View e può attuare proprietà aggiuntive che potrebbero non essere presenti sul Model. Si può anche applicare la convalida dei dati tramite le interfacce IDataErrorInfo o **INotifyDataErrorInfo**.
- Il ViewModel può definire stati logici che la View può rappresentare visivamente per l'utente.

La classe Model



Il Model nel pattern MVVM incapsula la logica di business e dati. Logica di business è definito come qualsiasi logica dell'applicazione che si occupa del recupero e della gestione dei dati e per fare in modo che siano imposti le regole di business che garantiscono la coerenza dei dati e la validità. Per massimizzare le opportunità di riutilizzo, i modelli non devono contenere alcun use case o user task.

In genere, il Model rappresenta il modello di dominio sul lato client dell'applicazione. Esso definisce le strutture dati basate sul modello di dati dell'applicazione, logica di business e di convalida. Il Model può includere anche il codice per l'accesso ai dati e il caching, anche se di solito è impiegato un repository di dati o servizio separato. Spesso, il Model e il Data Access Layer vengono generati come parte di una strategia di accesso ai dati o di servizio, come ad esempio il ADO.NET Entity Framework, WCF Data Services, o WCF RIA Services.

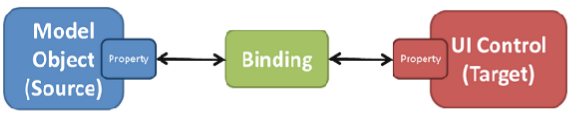
In genere, il Model implementa le strutture che lo rendono facile da associare alla View. Questo di solito significa che supporta le interfacce INotifyPropertyChanged e INotifyCollectionChanged. Le classi Model che rappresentano collezioni di oggetti in genere derivano dalla classe ObservableCollection<T>, che fornisce un'implementazione dell'interfaccia INotifyCollectionChanged.

Il Model può anche supportare la convalida dei dati e segnalazione degli errori attraverso le interfacce IDataErrorInfo (o INotifyDataErrorInfo). Essi consentono inoltre il supporto per la convalida dei dati e segnalazione degli errori all'interfaccia utente.

Il Model presenta le seguenti caratteristiche principali:

- Le classi del Model sono classi non visive che incapsulano i dati dell'applicazione e la logica di business. Esse sono responsabili della gestione dei dati dell'applicazione e garantire la coerenza e la validità incapsulando le regole di business necessari e logica di convalida dei dati.
- Le classi del Model non fanno riferimento direttamente alle classi View o ViewModel e non hanno alcuna dipendenza dalle modalità di attuazione.
- Le classi del Model in genere forniscono gli eventi cambiamenti attraverso le interfacce INotifyPropertyChanged e INotifyCollectionChanged. Questo permette loro di essere facilmente associati alla View. Classi del Model che rappresentano collezioni di oggetti in genere derivano dalla classe ObservableCollection<T>.
- Le classi del Model in genere forniscono la convalida dei dati e la segnalazione degli errori sia attraverso le interfacce IDataErrorInfo o INotifyDataErrorInfo.
- Le classi del Model sono tipicamente utilizzati in combinazione con un servizio o repository che incapsula l'accesso ai dati e la memorizzazione nella cache.

Data Binding



Associazione dei dati gioca un ruolo molto importante nel modello MVVM. Il ViewModel e (idealmente) il Model dovrebbero essere progettati per supportare questa associazione in modo che possano usufruire di queste funzionalità e quindi attuare le interfacce giuste.

Associazione dati WPF supporta modalità molteplici di associazione dati. Con l'associazione di dati a senso unico (One-Way), controlli dell'interfaccia utente possono essere legate ad un ViewModel in modo che riflettano il valore dei dati sottostanti quando viene eseguito il rendering. L'associazione bidirezionale dei dati (Two-Way) permette anche di aggiornare automaticamente i dati alla base alle modifiche nell'interfaccia utente.

Per garantire che l'interfaccia utente è tenuta aggiornata, questa dovrebbe implementare l'interfaccia di notifica di modifica: INotifyPropertyChanged, INotifyCollectionChanged o derivare dalla classe ObservableCollection<T> (che fornisce un'implementazione di questa interfaccia).

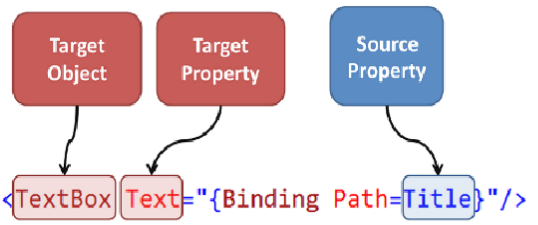
Nei casi dove un ViewModel restituisce oggetti nidificate tramite la proprietà Path, tutti i ViewModel e Model accessibili alla View dovrebbero implementare le interfacce INotifyPropertyChanged e/o INotifyCollectionChanged.

Implementazione INotifyPropertyChanged

Implementare l'interfaccia INotifyPropertyChanged nella ViewModel o Model permette loro di fornire le notifiche di modifica a tutti i controlli con associazione a dati nella View quando il valore della proprietà sottostante cambia. L'implementazione di questa interfaccia è semplice, e viene mostrato nel seguente esempio.


```
    public class ColoreViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private Colore model;

        public double Red
        {
            get { return model.Rosso; }
            set {
                  if (model.Rosso != (byte)value)
                  {
                      model.Rosso = (byte)value;
                      OnPropertyChanged("Red");
                  }
            }
        }
...
        protected void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
                handler(this, new PropertyChangedEventArgs(propertyName));
        }
```



Implementare l'interfaccia INotifyPropertyChanged su molte ViewModel può essere ripetitivo e soggetto ad errori a causa della necessità di specificare il nome della proprietà nell'argomento evento. È auspicabile definire la classe base BindableBase da cui è possibile derivare i tuoi visualizzazione classi del modello che implementa l'interfaccia INotifyPropertyChanged in modo type-safe.

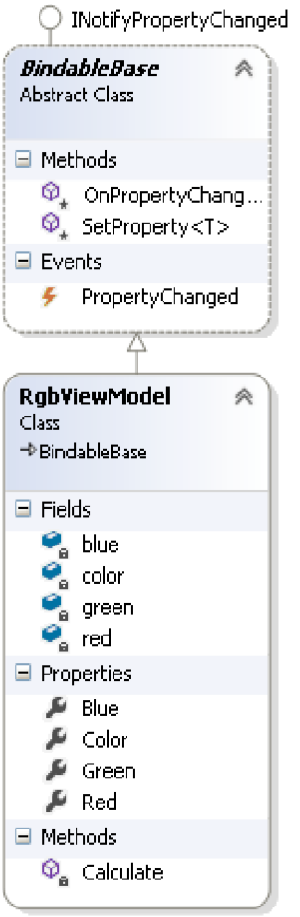
```
    public abstract class BindableBase : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged = delegate { };

        protected void OnPropertyChanged(string propertyName)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }

        protected virtual void SetProperty<T>(ref T member, T value, [CallerMemberName] string propertyName = null)
        {
            if (object.Equals(member, value))
                return;
            member = value;
            OnPropertyChanged(propertyName);
        }

        protected void OnPropertyChanged<T>(Expression<Func<T>> propertyExpression)
        {
            var body = propertyExpression.Body as MemberExpression;
            var expression = body.Expression as ConstantExpression;
            PropertyChanged(expression.Value, new PropertyChangedEventArgs(body.Member.Name));
        }
    }
```

In questo modo si ha a disposizione un superclasse che si preoccupa di gestire tutta la parte di notifica.




```
public class ColoreViewModel : BindableBase
{

    private Colore model;

    public double Red
    {
        get { return model.Rosso; }
        set {
              if (model.Rosso != (byte)value)
              {
                  model.Rosso = (byte)value;
                  OnPropertyChanged("Red");
                  OnPropertyChanged(() => Red);
                  //OnPropertyChanged(() => Blu);

                  //double red = model.Rosso;
                  //SetProperty<double>(ref red, value);
                  //model.Rosso = (byte)red;
              }
        }
    }
```

Se il Model o il ViewModel include proprietà i cui valori influiscono anche su altre proprietà, bisogna assicurarsi di sollevare gli eventi di notifica per tutte le proprietà (esempio con blue, nel codice commentato).

ObservableCollection<T>

La View o il ViewModel può rappresentare una collezione di oggetti o definire una o più proprietà che restituiscono un insieme di elementi. In entrambi i casi, è probabile che si vuole visualizzare la collezione in un ItemsControl, ad esempio un ListBox, o in un controllo DataGrid nella View. Questi controlli possono essere associati tramite la proprietà ItemSource, che rappresenta una raccolta di un insieme, a un Model.

Per supportare adeguatamente le richieste di notifica di modifica, il ViewModel o il Model, deve implementare l'interfaccia INotifyCollectionChanged (oltre all'interfaccia INotifyPropertyChanged). Se il modello di visualizzazione o classe del modello definisce una proprietà che restituisce un riferimento a un insieme, la classe insieme restituita deve implementare l'interfaccia INotifyCollectionChanged. Invece di implementare direttamente l'interfaccia, è spesso più facile usare o derivare dalla classe ObservableCollection<T> che fornisce un'implementazione di questa interfaccia ed è comunemente utilizzata sia come classe base sia per implementare le proprietà che rappresentano un insieme di elementi.

```
public class OrderViewModel : BindableBase
{
    public ObservableCollection<OrderLineItem> LineItems { get; private set; }

    public OrderViewModel(IOrderService orderService)
    {
        LineItems = new ObservableCollection<OrderLineItem>(
                        orderService.GetLineItemList());
    }
}
```

Costruzione e Collegamenti

Il modello MVVM consente di separare in modo pulito l'interfaccia utente e la logica e i dati di presentazione e di business. Gestire le interazioni tra View e ViewModel attraverso le classi di associazione dei dati e comandi sono altri aspetti importanti da considerare. Il passo successivo è quello di considerare come la View, il ViewModel, e le classi del Model sono istanziati e associati tra loro in fase di esecuzione.

Tipicamente, vi è una relazione uno-a-uno tra una View e il suo ViewModel (le finestre MDI non sono più usate frequentemente, anche se per esempio vengono usate in Outlook quando si crea un nuovo messaggio). Bisogna scegliere come gestire l'istanza della View e del ViewModel e la loro associazione tramite la proprietà DataContext in fase di esecuzione.

Ci sono diversi modi come la View e il ViewModel possono essere costruiti e associati in fase di esecuzione. L'approccio più appropriato dipende dal fatto se si crea la View o il ViewModel prima, e se si esegue questa operazione a livello di programmazione o dichiarativamente. Le sezioni seguenti descrivono modi comuni in cui la View e ViewModel possono essere creati e collegati tra loro in fase di esecuzione.

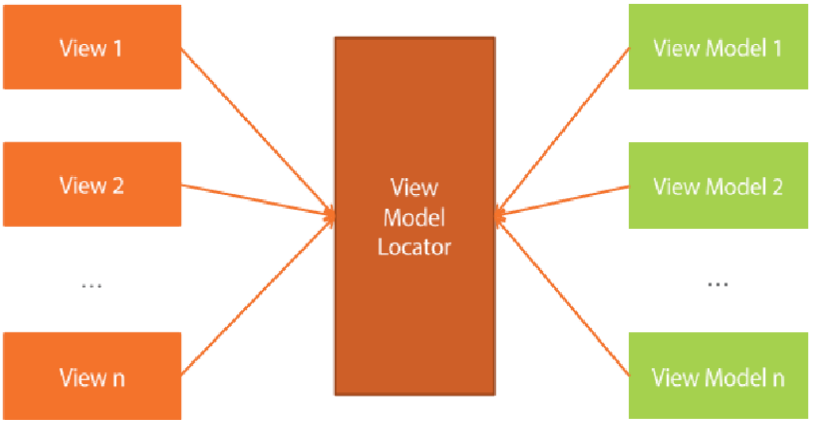
View First

```
<Window.DataContext>
    <viewmodels:CoffeeOverviewViewModel>
    </viewmodels:CoffeeOverviewViewModel>
</Window.DataContext>
```

Normalmente si sceglie questo approccio, che tuttavia non può essere utilizzato se le View vengono create dinamicamente (applicazioni MDI come Outlook). In questo caso è opportuno utilizzare un ViewModel Locator.

ViewModel Locator (con IoC)

È una classe che conosce tutti i ViewModel dell'applicazione. Tutte le View conoscono il ViewModel Locator e chiedono di poter ritornare un determinato ViewModel.



La versione manuale obbliga la definizione e istanziazione di tutti i ViewModel nel ViewModelLocator, mentre per migliorare la getione, è consigliabile utilizzare il pattern Inversion of Control (IoC), il quale permette la creazione su "richiesta" della classe (o interfaccia) necessaria. In questo casi viene usato SimpleIoc (che necessita del pacchetto NuGet Microsoft.Practices.ServiceLocation).

```
public ViewModelLocator()
{
    ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

    SimpleIoc.Default.Register<CoffeeDataRepository>();
    // se interfaccia SimpleIoc.Default.Register<ICoffeeDataRepository, CoffeeDataRepository>();

    SimpleIoc.Default.Register<MainViewModel>();
    SimpleIoc.Default.Register<AboutViewModel>();
    SimpleIoc.Default.Register<CoffeeListViewModel>();
    SimpleIoc.Default.Register<CoffeeDetailViewModel>(true); // per forzare la creazione subito (messaggio)
}
```

Vengono definiti i vari ViewModel in modo statico che potranno essere usati come proprietà.

```
public static MainViewModel Main { get { return ServiceLocator.Current.GetInstance<MainViewModel>(); } }

public static CoffeeListViewModel CoffeeList
                            { get { return ServiceLocator.Current.GetInstance<CoffeeListViewModel>(); } }
```


Questi deve essere a sua volta essere istanziato in Appl.xaml nel <ResourceDictionary> con un x:Key mentre nella View si definisce nell'header un DataContext indicando di quale ViewModel si necessita.

```
In App.xaml
    <Application.Resources>
        <ResourceDictionary>
            <local:ViewModelLocator x:Key="Locator"></local:ViewModelLocator>


In View
<UserControl x:Class="_08_CoffeeShop.View.CoffeeListView"
...
             DataContext="{Binding Source={StaticResource Locator}, Path=CoffeeList}"
```

In questo esempio il CoffeeListViewModel registrerà un messaggio (vedi comunicazione tra ViewModel) per mostrare la propria View:

```
public CoffeeDetailViewModel()
{
    Messenger.Default.Register<Coffee>(this, OnSentCoffee);
}

private void OnSentCoffee(Coffee obj)
{
    SelectedCoffee = obj;
    ViewModelLocator.Main.CurrentViewModel = this;
}
```

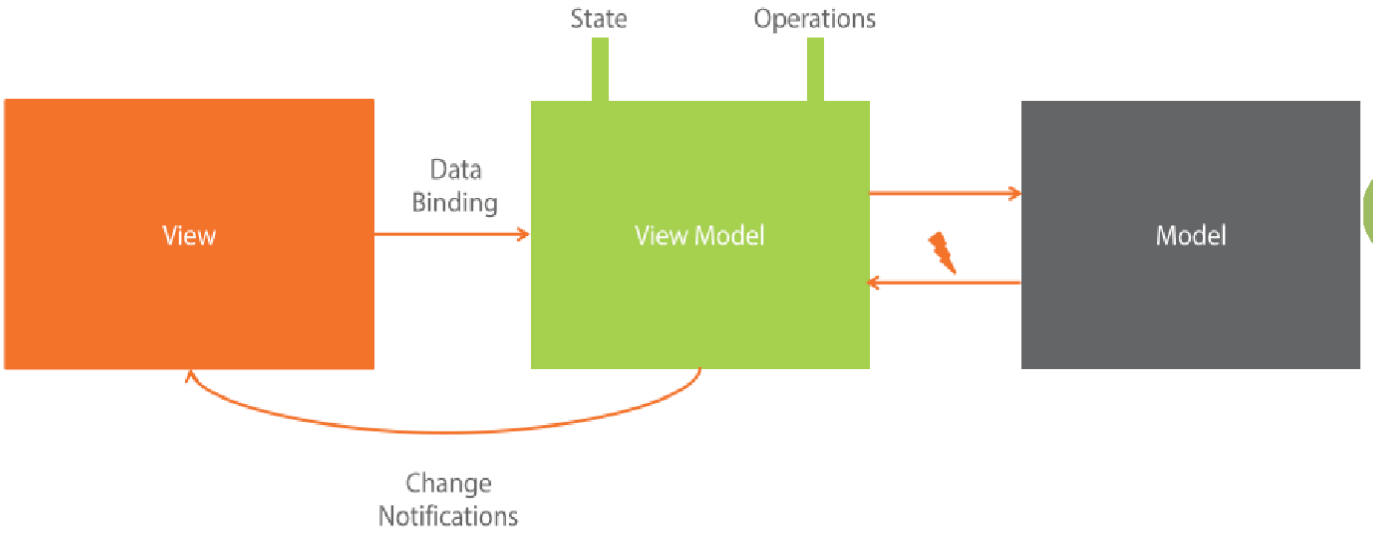
ViewModel First

```
public CoffeeOverviewViewModel(CoffeeOverviewView view)
{
     view.ViewModel = this;
}
```

In questo caso il ViewModel viene creato per prima e con il parametro della View viene assegnata la proprietà ViewModel con il ViewModel stesso.

Comandi

Oltre a fornire l'accesso ai dati e la loro modifica nella View, il ViewModel definisce normalmente una o più azioni od operazioni che possono essere eseguite dall'utente, in genere definiti come comandi. I comandi forniscono un modo per rappresentare azioni od operazioni che possono essere facilmente legati ai controlli nell'interfaccia utente. Essi racchiudono il codice che implementa l'azione e contribuire a mantenerlo disaccoppiato nella View.



I comandi possono essere rappresentati visivamente e invocati in modi diversi dall'utente quando interagiscono con la View. Nella maggior parte dei casi sono invocati con un clic del mouse, ma possono anche essere invocati da altri eventi di input. L'interazione tra l'UI e il comando può essere bidirezionale. In questo caso, il comando può essere invocato quando l'utente interagisce con l'interfaccia utente e l'interfaccia utente può abilitare o disabilitare automaticamente il comando sottostante.

Il ViewModel implementa i comandi o come metodo di comando (non sviluppato in questo documento) o come oggetto Command (un oggetto che implementa l'interfaccia ICommand). Ad esempio, alcuni controlli in WPF intrinsecamente supportano comandi e forniscono una proprietà di comando che possono essere dati associati a un oggetto ICommand fornito dal modello vista. In altri casi, un comportamento (behavior) può essere utilizzato per associare un controllo con un oggetto metodo comando fornito dal ViewModel.

Implementazione di Oggetti Command

Un oggetto Command è un oggetto che implementa l'interfaccia ICommand. Questa interfaccia definisce un metodo Execute, che incapsula l'operazione stessa, e un metodo CanExecute, che indica se il comando può essere richiamato in un particolare momento.

La classe DelegateCommand incapsula due delegati che fanno riferimento a metodo implementato all'interno della classe ViewModel. Essa implementa i metodi Execute e CanExecute dell'interfaccia ICommand invocando questi delegati. È possibile specificare i delegati per i metodi di ViewModel nel costruttore della classe DelegateCommand.

Ad esempio, il codice seguente istanzia DelegateCommand, che rappresentano due comandi, specificando i delegati OnSubmit (OnStart e OnStop) e CanSubmit (CanStart e CanStop), metodi del ViewModel. I comandi vengono esposti alla View tramite una proprietà di sola lettura che restituisce un riferimento a un ICommand (IDelegateCommand).

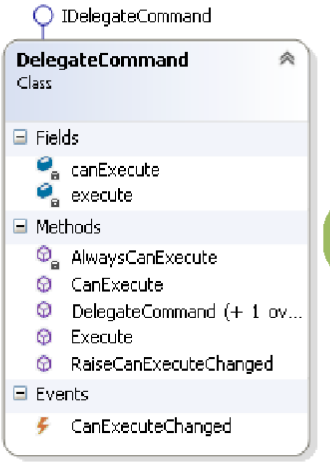
```
        #region ================== membri ====================
        public IDelegateCommand StartCommand { get; protected set; }
        public IDelegateCommand StopCommand { get; protected set; }
        #region ================== costruttori ===============
        public CronometroViewModel()
        {
            RegisterCommands();

            model = new Cronometro();
...
        private void RegisterCommands()
        {
            StartCommand = new DelegateCommand(OnStart, CanStart);
            StopCommand = new DelegateCommand(OnStop, CanStop);
        }
...
        #region ================== metodi aiuto ============
        private void OnStart(object obj)
        {
            Start();
            StartCommand.RaiseCanExecuteChanged();
            StopCommand.RaiseCanExecuteChanged();
        }

        private bool CanStart(object arg)
        {
            //return true;
            return !model.IsRunning;
        }

        private void OnStop(object obj)
        {
            Stop();
            StartCommand.RaiseCanExecuteChanged();
            StopCommand.RaiseCanExecuteChanged();
        }

        private bool CanStop(object arg)
        {
            //return true;
            return model.IsRunning;
        }
```



Quando il metodo Execute viene chiamato sull'oggetto DelegateCommand, inoltra semplicemente la chiamata al metodo del ViewModel tramite il delegato specificato nel costruttore. Allo stesso modo, quando il metodo CanExecute viene chiamato, il metodo corrispondente viene eseguito. Il delegato al metodo CanExecute nel costruttore è facoltativo. Se un delegato non è specificato, restituirà sempre true.

Per poter sfruttare attivo/disattivo del Button si deve richiamare al posto corretto il metodo RaiseCanExecuteChanged nelle parti di codice che modificano l possibilità di utilizzo del comando (in questo caso quando parte il cronometro e quando si ferma.

Il ViewModel può indicare un cambiamento di stato del comando CanExecute chiamando il metodo RaiseCanExecuteChanged. Ciò solleva l'evento CanExecuteChanged. I controlli dell'interfaccia utente che sono legati al comando aggiorneranno il loro stato di abilitazione riflettendo la disponibilità del comando associato.

Sono disponibili altre implementazioni dell'interfaccia ICommand. La classe ActionCommand fornita dal Expression Blend SDK è simile alla classe DelegateCommand di Prism descritto in precedenza, ma supporta solo un singolo metodo Execute delegato.

Invocare Comando Oggetti da View

Alcuni controlli WPF, in particolare i controlli derivati da ButtonBase (Button o RadioButton), collegamenti ipertestuali o controlli derivati da MenuItem, possono essere facilmente dati associati ad un oggetto di comando tramite la proprietà Command.

```
<Button Command="{Binding StartCommand}">Start</Button>
<Button Command="{Binding StopCommand}">Stop</Button>
```

Attached Properties / Behaviors

I comandi si possono usare solo su controlli che derivano da ButtonBase (Button, RadioButton, CheckBox, ...) e possono essere usati solo per alcuni eventi (es non funziona 2xClick)

I Behavior permettono di racchiudere delle funzionalità in un componente riutilizzabile che possono essere "attaccati" ad un componente permettendo di estendere i comportamenti (Behavior) degli oggetti senza modificare l'oggetto originale (qualcosa di simile all'estensione di metodi) XAML.

Ci sono 2 opzioni: Attached behaviour e blend behaviour (dalla libreria Microsoft.Interactivity)

Per evitare problemi di errori e per una maggiore chiarezza di codice si usa normalmente la seconda opzione.

Per poter usufruire dei behavior è necessario operare come segue:

- Nelle referenze ci devono essere System.Windows.Interactivity e Microsoft.Expression.Interactions
- Aggiungere i seguenti namespace

```
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
```

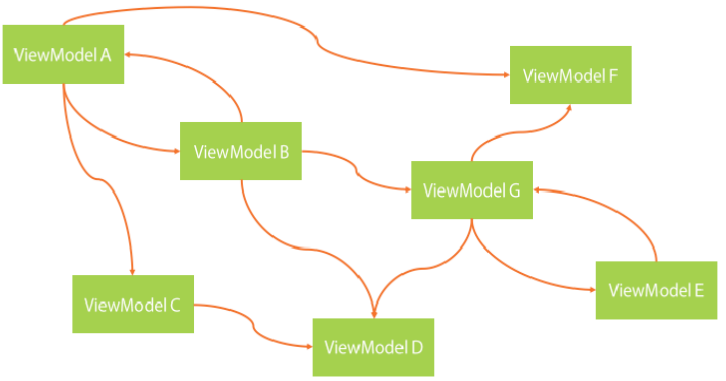
- Usare uno dei codici seguenti per assegnare a qualsiasi EventName il comando da eseguire

Con CallMethodAction	Con InvokeCommandAction

`<i:Interaction.Triggers>`
`  <i:EventTrigger EventName="Click">`
`  <ei:CallMethodAction TargetObject="{Binding}"`
`  MethodName="SaveCommand"/>`
`  </i:EventTrigger>`
`</i:Interaction.Triggers>`
	`<i:Interaction.Triggers>`
	`  <i:EventTrigger EventName="Click">`
	`  <i:InvokeCommandAction`
	`  Command="{Binding SaveCommand}"/>`
	`  </i:EventTrigger>`
	`</i:Interaction.Triggers>`

Comunicazione tra ViewModel

Un aspetto importante è la comunicazione tra i vari ViewModel per passare informazioni tra le View senza avere alcun HardLink.



Per risolvere questo problema ci si affida a un Messenger (o Mediator). Si trova in mezzo nella comunicazione tra più ViewModels che tutti conoscono e comunicano attraverso lui.

Per esempio si registra ViewModelA con il Messenger e si definisce che potrebbe inviare un messaggio di un certo tipo. ViewModelB e ViewModelC sono interessati a quello che "dice" ViewModelA. Questi "informano" il Messenger che sono interessati ad un certo tipo di messaggio e non appena riceve un messaggio del tipo coinvolto, questi lo invierà agli interessati.

Se interessa inviare un valore/oggetto a chiunque sia interessato basta inviare il messaggio.

```
private void EditCoffee(object obj)
{
    Messenger.Default.Send<Coffee>(selectedCoffee);
```

Il destinatario deve aver registrato il suo interessamento a quel determinato tipo di messaggio. Questo avviene (per convenzione) nel metodo RegisterMessages, richiamato dal costruttore del ViewModel, dove viene indicato quale metodo deve essere eseguito.

```
private void RegisterMessages()
{
    Messenger.Default.Register<Coffee>(this, OnCoffeeReceived);
}

public void OnCoffeeReceived(Coffee coffee)
{
    SelectedCoffee = coffee;
}
```

Se invece si vuole inviare un messaggio particolare, in questo caso per un aggiornamento e salvataggio, si può definire un altro messaggio (classe), che anche se vuoto, viene ricevuto e elaborato dal ViewModel interessato.

```
public class UpdateListMessage
{
}
```

Nel codice del ViewModel si invia il messaggio quando è il caso

```
private void DeleteCoffee(object coffee)
{
    Messenger.Default.Send<UpdateListMessage>(new UpdateListMessage());
}
```

```
private void SaveCoffee(object coffee)
{
    Messenger.Default.Send<UpdateListMessage>(new UpdateListMessage());
}
```

E nel ViewModel interessato si esegue lo stesso approccio visto prima.

```
private void RegisterMessages()
{
    Messenger.Default.Register<UpdateListMessage>(this, OnUpdateListMessageReceived);
}

private void OnUpdateListMessageReceived(UpdateListMessage obj)
{
    LoadData();
}
```


Convalida dei dati e Segnalazione errori

È spesso necessario eseguire la convalida dei dati e segnalare eventuali errori alla View in modo che l'utente può intervenire per correggerli.

Anche se WPF fornisce il supporto per la gestione di errori, questo modo dovrebbe essere evitato se possibile. Un approccio alternativo è quello di implementare le interfacce IDataErrorInfo o INotifyDataErrorInfo sulle classi ViewModel o Model. Queste interfacce consentono al ViewModel o Model di eseguire la convalida dei dati per uno o più valori di proprietà e restituire un messaggio di errore alla View in modo da avvisare l'utente dell'errore.

Implementazione IDataErrorInfo

L'interfaccia IDataErrorInfo fornisce il supporto base per la convalida dei dati di proprietà e la segnalazione degli errori. Esso definisce due proprietà di sola lettura: una proprietà indicizzatore, con il nome della proprietà come argomento indicizzatore, e una proprietà Error. Entrambe le proprietà restituiscono un valore stringa.

Implementazione INotifyDataErrorInfo

L'interfaccia INotifyDataErrorInfo è più flessibile dell'interfaccia IDataErrorInfo. Essa supporta più errori per proprietà, la convalida asincrona dei dati, e la capacità di comunicare alla View se cambia lo stato di errore.

Essa definisce una proprietà HasErrors, che consente al ViewModel di indicare se un errore (o più errori) esiste, e un metodo GetErrors, che consente al ViewModel di restituire un elenco di errori per una particolare proprietà.

L'interfaccia INotifyDataErrorInfo definisce anche un evento ErrorsChanged. Questo supporta gli scenari di convalida asincrone, consentendo la View o ViewModel di segnalare un cambiamento di stato di errore per una particolare proprietà attraverso l'evento ErrorsChanged. L'evento ErrorsChanged permette al ViewModel di informare la View di un errore una volta che è stato identificato un errore nella convalida dei dati.

Per supportare INotifyDataErrorInfo, è necessario mantenere un elenco di errori per ogni proprietà. Vedi il codice della classe ValidatableBindableBase dove un classe Dictionary tiene traccia di tutti gli errori di convalida nell'oggetto. Notifica se la lista degli errori cambia.

Quando si associa controlli nella visualizzazione di proprietà che si desidera validare attraverso l'interfaccia IDataErrorInfo, impostare la proprietà ValidatesOnDataErrors a true del Binding.

Es:

```
Text="{Binding Path=LastName, ValidatesOnNotifyDataErrors=True}"
```

Per poter agire anche sui Command (abilitare/disabilitare Button) bisogna assegnare l'evento se ci sono modifiche negli errori dove, in questo caso, il Save viene disabilitato se ci sono errori.

```
        public void SetCustomer(Customer cust)
        {
            _editingCustomer = cust;
            if (Customer != null) Customer.ErrorsChanged -= RaiseCanExecuteChanged;
            Customer = new SimpleEditableCustomer();
            Customer.ErrorsChanged += RaiseCanExecuteChanged;
            CopyCustomer(cust, Customer);
        }

        private void RaiseCanExecuteChanged(object sender, EventArgs e)
        {
            SaveCommand.RaiseCanExecuteChanged();
        }

        private bool CanSave()
        {
            return !Customer.HasErrors;
        }
```


Servizi

Vari aspetti non sono presi in considerazione da parte di MVVM come la navigazione da una schermata all'altra, come aprire una finestra di dialogo (per esempio per mostrare errori; non è compito del ViewModel di mostrare un dialogo ma solamente sapere che è necessario mostrarlo - per questo c'è la classe DialogService), come accedere al Model/Dati (repository) e altre attività non coperte finora.

Servizi sono semplici classi con una particolare funzionalità e vengono registrati di solito a livello di applicazione (IOC container)

Navigazione

Per consentire la navigazione tra una pagina e l'altra bisogna considerare l'approccio dell'applicazione rispetto alle direzioni guida sull'interfaccia. Se si desidera aprire fisicamente un'altra finestra si presuppone che l'operazione è relativamente complessa e non è compito del ViewModel di affrontare questo compito. Per questo caso si crea un servizio (nella cartella Services) che si occupa della creazione e della chiusura di una finestra specifica, mentre la gestione dello stesso avviene nel ViewModel (vedi nella sezione implementazione DialogCoffeeDialogService).

L'approccio più vicino allo sviluppo con il pattern MVVM è la creazione di un MainViewModel che gestisce i vari ViewModel e il Command (con il Parameter) o sottoscrivere un messaggio per selezionare le relative maschere (vedi nella sezione implementazione MainViewModel).

```
    <UserControl.Resources>
        <DataTemplate DataType="{x:Type viewmodel:AboutViewModel}">
            <view:AboutView/>
        </DataTemplate>
    </UserControl.Resources>
...
        <ContentControl Grid.Row="1" Content="{Binding Path=CurrentViewModel}"/>
```

Ogni DataTemplate mappa il DataType di un ViewModel alla View. L'uso di un determinato DataTemplate crea la View e assegna il DataContext al ViewModel. Ogniqualvolta che si assegna a CurrentViewModel uno dei ViewModel,

viene mostrata la rispettiva View. È necessario nel MainViewModel una proprietà CurrentViewModel, della logica e del comanding per scambiare il riferimento tra i vari ViewModel

CurrentViewModel è il padre nella gerarchia dei ViewModel e serve per la navigazione. I figli sono gestiti da questo padre. Il ContentControl nella MainWindow agisce come contenitore delle View da visualizzare; l'operazione sui ViewModel viene fatta dal CurrentViewModel.

Visto che il CurrentViewModel viene assegnato più volte a dipendenza della navigazione, la classe deve implementare INPC, come pure tutti i ViewModel.

Data Access

In Service viene definito il servizio DataRepository. Per ogni sorgente dati viene definito il proprio Repository in modo da rendere uniforme l'accesso ai dati usando sempre gli stessi metodi, qualsiasi fonte dati sia. Questo permette di occuparsi unicamente della gestione dei dati senza doversi preoccupare come effettuare le varie operazioni CRUD.