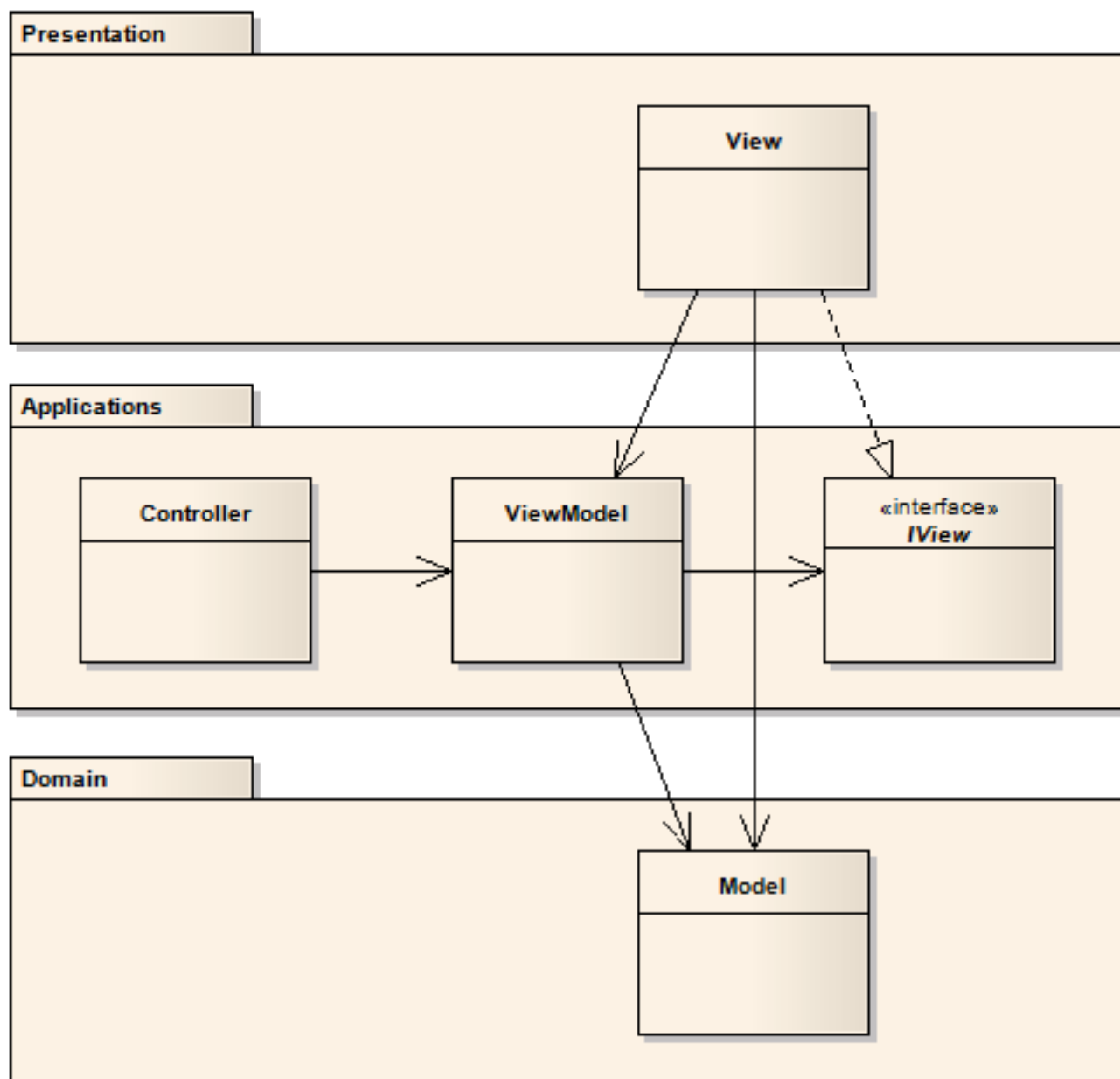


دز طراحی پروژه‌های مقیاس بزرگ و البته به صورت ماژولار همیشه ساختار پروژه اهمیت به سزایی دارد. متأسفانه این مورد خیلی در طراحی پروژه‌ها در نظر گرفته نمی‌شود و اغلب اوقات شاهد آن هستیم که یک پروژه بسیار بزرگ دقیقاً به همان صورت پروژه‌های کوچک و کم اهمیت‌تر مدیریت و پیاده سازی می‌شود که این مورد هم مربوط به پروژه‌های تحت وب و هم پروژه‌های تحت ویندوز و WPF است. برای مدیریت پروژه‌های WPF و Silverlight در این [پست](#) به اختصار درباره PRISM بحث شد. مزایا و معایب آن بررسی و در طی این پست‌ها ( [^](#) و [^](#) ) مثال هایی را پیاده سازی کردیم. اما در این پست مفتخرم شما را با یکی دیگر از کتابخانه‌های مربوط به پیاده سازی مدل MVVM آشنا کنم. کتابخانه ای متن باز، بسیار سبک با کارایی بالا. اما نکته ای که ذکر آن خالی از لطف نیست این است که قبلاً از این کتابخانه در یک پروژه بزرگ و ماژولار WPF استفاده کردم و نتیجه مطلوب نیز حاصل شد.

معرفی:

WPF Application Framework یا به اختصار WAF کتابخانه کم حجم سبک و البته با کارایی عالی برای طراحی پروژه‌های ماژولار WPF در مقیاس بزرگ طراحی شده است که مدل پیاده سازی آن بر مبنای مدل MVVM و MVC است. شاید برایتان جالب باشد که این کتابخانه دقیقاً مدل MVC را با مدل MVVM ترکیب کرده در نتیجه مفاهیم آن بسیار شبیه به پروژه‌های تحت وب MVC است. همانطور که از نام آن پیداست این کتابخانه صرفاً برای پروژه‌های WPF طراحی شده، در نتیجه در پروژه‌های Silverlight نمی‌توان از آن استفاده کرد.

ساختار کلی آن به شکل زیر می‌باشد:



همانطور که مشاهده می‌کنید پروژه‌های مبتنی بر این کتابخانه همانند سایر کتابخانه‌های MVVM از سه بخش تشکیل شده اند. بخش اول با عنوان Shell یا Presentation معرف فایل‌های Xaml پروژه است، بخش دوم یا Application معرف ViewModel و Controller و البته IView می‌باشد. بخش Domain نیز در برگیرنده مدل‌های برنامه است.

#### معرفی برخی مفاهیم:

«Shell»: این کلاس معادل یک فایل Xaml است که حتما باید یک اینترفیس IView را پیاده سازی نماید.

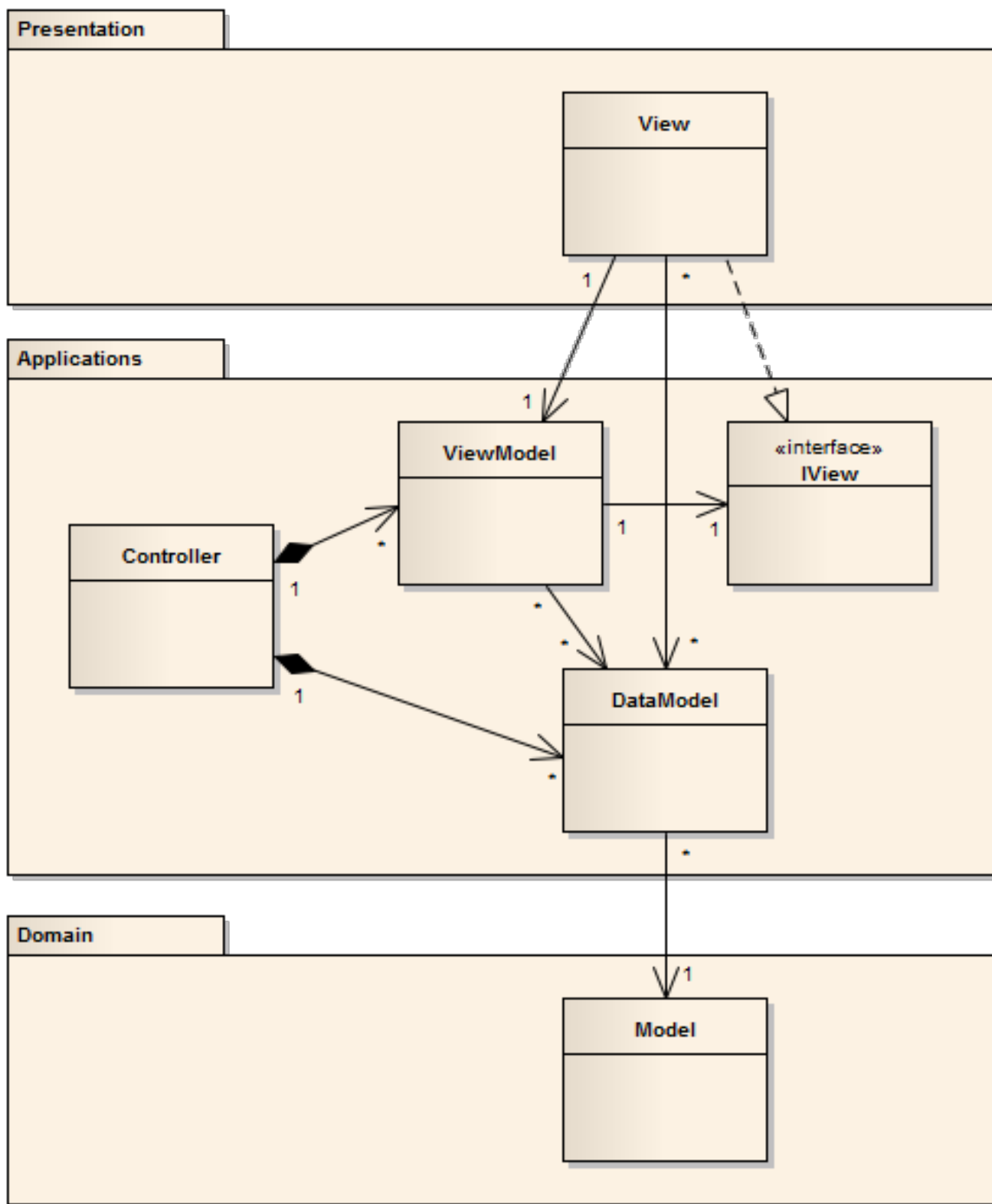
«IView»: معرف یک اینترفیس جهت برقراری ارتباط بین ViewModel و Shell

«ViewModel»: در این جا ViewModel با مفهوم ViewModel در سایر کتابخانه‌های MVVM کمی متفاوت است. در این کتابخانه ViewModel فقط شامل تعاریف است و هیچ گونه پیاده سازی در اینجا صورت نمی‌گیرد. دقیقا معادل مفهوم ViewModel در پروژه‌های MVC تحت وب.

«Controller»: پیاده سازی ViewModel و تعریف رفتارها در این قسمت انجام می‌گیرد.

اما در بسیاری از پروژه‌ها نیاز به پیاده سازی الگوی DataModel-View-ViewModel است که این کتابخانه با در اختیار داشتن برخی

کلاس‌های پایه این مهم را برایمان میسر کرده است.



همانطور که می‌بینید در این حالت بر خلاف حالت قبلی **ViewModel** و کنترلرهای پروژه به جای ارتباط با مدل با مفهوم **DataModel** تغذیه می‌شوند که یک پیاده سازی سفارشی از مدل‌های پروژه است. هم چنین این کتابخانه یک سری **Converter**های سفارشی

جهت تبدیل Model به DataModel و برعکس را ارائه می‌دهد. سرویس‌های پیش فرض: که شامل DialogBox جهت نمایش پیام‌ها و Save|Open File Dialog سفارشی نیز می‌باشد. «برای پیاده سازی Modularity از کتابخانه MEF استفاده شده است. Command های سفارشی: پیاده سازی خاص از اینترفیس ICommand «مفاهیم مربوط به [Weak Event Pattern](#) به صورت توکار در این کتابخانه تعبیه شده است. «به صورت پیش فرض مباحث مربوط به اعتبارسنجی با استفاده از [DataAnnotation](#) و [IDataErrorInfo](#) در این کتابخانه تعبیه شده است. «ارائه Extension های مربوط به UnitTest نظیر Exceptions و CanExecuteChangedEvent و PopertyChanged جهت سهولت در تهیه unit test

دانلود و نصب

با استفاده از nuget و دستور زیر می‌توانید این کتابخانه را نصب نمایید:

```
Install-Package waf
```

هم چنین می‌توانید سورس آن به همراه فایل‌های باینری را از [اینجا](#) دریافت کنید. در پست بعدی یک نمونه از پیاده سازی مثال با این کتابخانه را بررسی خواهیم کرد.

در این [پست](#) با مفاهیم اولیه این کتابخانه آشنا شدید. برای بررسی و پیاده سازی مثال، ابتدا یک Blank Solution را ایجاد نمایید. فرض کنید قصد پیاده سازی یک پروژه بزرگ ماژولار را داریم. برای این کار لازم است مراحل زیر را برای طراحی ساختار مناسب پروژه دنبال نمایید.

**نکته:** آشنایی اولیه با مفاهیم [MEF](#) از ملزومات این بخش است.

«ابتدا یک Class Library به نام Views ایجاد نمایید و اینترفیس زیر را به صورت زیر در آن تعریف نمایید. این اینترفیس رابط بین کنترلر و View از طریق ViewModel خواهد بود.

```
public interface IBookView : IView
{
    void Show();
    void Close();
}
```

اینترفیس IView در مسیر System.Waf.Applications قرار دارد. در نتیجه از طریق nuget اقدام به نصب Package زیر نمایید:

Install-Package WAF

«حال در Solution ساخته شده یک پروژه از نوع WPF Application به نام Shell ایجاد کنید. با استفاده از نیوگت، Waf Package را نصب نمایید؛ سپس ارجاعی از اسمبلی Views را به آن ایجاد کنید. output type اسمبلی Shell را به نوع ClassLibrary تغییر داده، همچنین فایل‌های موجود در آن را حذف نمایید. یک فایل Xaml جدید را به نام BookShell ایجاد نمایید و کدهای زیر را در آن کپی نمایید:

```
<Window x:Class="Shell.BookShell"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Book View" Height="350" Width="525">
    <Grid>
        <DataGrid ItemsSource="{Binding Books}" HorizontalAlignment="Left" Margin="10,10,0,0"
        VerticalAlignment="Top" Width="400" Height="200">
            <DataGrid.Columns>
                <DataGridTextColumn Header="Code" Binding="{Binding Code}"
                Width="100"></DataGridTextColumn>
                <DataGridTextColumn Header="Title" Binding="{Binding Title}"
                Width="300"></DataGridTextColumn>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

این فرم فقط شامل یک دیتاگرید برای نمایش اطلاعات کتاب‌هاست. دیتای آن از طریق ViewModel تامین خواهد شد، در نتیجه ItemsSource آن به خاصیتی به نام Books باید شده است. حال ارجاعی به اسمبلی System.ComponentModel.Composition دهید. سپس در Code behind این فرم کدهای زیر را کپی کنید:

```
[Export(typeof(IBookView))]
[PartCreationPolicy(CreationPolicy.NonShared)]
public partial class BookShell : Window, IBookView
{
    public BookShell()
    {
        InitializeComponent();
    }
}
```

کاملاً واضح است که این فرم اینترفیس IBookView را پیاده سازی کرده است. از آنجاکه کلاس Window به صورت پیش فرض دارای متدهای Show و Close است در نتیجه نیازی به پیاده سازی مجدد متدهای IBookView نیست. دستور Export باعث می‌شود که این کلاس به عنوان وابستگی به Composition Container اضافه شود تا در جای مناسب بتوان از آن وهله سازی کرد. نکته‌ی مهم این است که به دلیل آنکه این کلاس، اینترفیس IBookView را پیاده سازی کرده است در نتیجه نوع Export این کلاس حتماً باید به صورت صریح از نوع IBookView باشد.

«یک Class Library به نام Models بسازید و بعد از ایجاد آن، کلاس زیر را به عنوان مدل Book در آن کپی کنید:

```
public class Book
{
    public int Code { get; set; }

    public string Title { get; set; }
}
```

«یک Class Library دیگر به نام ViewModels ایجاد کنید و همانند مراحل قبلی، Package مربوط به WAF را نصب کنید. سپس کلاسی به نام BookViewModel ایجاد نمایید و کدهای زیر را در آن کپی کنید (ارجاع به اسمبلی‌های Views و Models را فراموش نکنید):

```
[Export]
[Export(typeof(ViewModel<IBookView>))]
public class BookViewModel : ViewModel<IBookView>
{
    [ImportingConstructor]
    public BookViewModel(IBookView view)
        : base(view)
    {
    }

    public ObservableCollection<Book> Books { get; set; }
}
```

ViewModel مورد نظر از کلاس ViewModel of T ارث برده است. نوع این کلاس معادل نوع View مورد نظر ماست که در اینجا مقصود IBookView است. این کلاس شامل خاصیتی به نام ViewCore است که امکان فراخوانی متدها و خاصیت‌های View را فراهم می‌نماید. وظیفه اصلی کلاس پایه ViewModel، وهله سازی از View سپس ست کردن خاصیت DataContext در View مورد نظر به نمونه وهله سازی شده از ViewModel است. در نتیجه عملیات مقید سازی در Shell به درستی انجام خواهد شد. به دلیل اینکه سازنده پیش فرض در این کلاس وجود ندارد حتماً باید از ImportingConstructor استفاده نماییم تا CompositionContainer در هنگام عملیات وهله سازی Exception صادر نکند.

«بخش بعدی ساخت یک Class Library دیگر به نام Controllers است. در این Library نیز بعد از ارجاع به اسمبلی‌های زیر کتابخانه WAF را نصب نمایید.

Views

Models

ViewModels

System.ComponentModel.Composition

کلاسی به نام BookController بسازید و کدهای زیر را در آن کپی نمایید:

```
[Export]
public class BookController
{
    [ImportingConstructor]
    public BookController(BookViewModel viewModel)
    {
        ViewModelCore = viewModel;
    }

    public BookViewModel ViewModelCore
    {
    }
```

```

        get;
        private set;
    }

    public void Run()
    {
        var result = new List<Book>();
        result.Add(new Book { Code = 1, Title = "Book1" });
        result.Add(new Book { Code = 2, Title = "Book2" });
        result.Add(new Book { Code = 3, Title = "Book3" });

        ViewModelCore.Books = new ObservableCollection<Models.Book>(result);

        (ViewModelCore.View as IBookView).Show();
    }
}

```

نکته مهم این کلاس این است که BookViewModel به عنوان وابستگی این کنترلر تعریف شده است. در نتیجه در هنگام و هله سازی از این کنترلر Container مورد نظر یک و هله از BookViewModel را در اختیار آن قرار خواهد داد. در متد Run نیز ابتدا مقدار Book که به ItemsSource دیتا گرید در BookShell مقید شده است مقدار خواهد گرفت. سپس با فراخوانی متد Show از اینترفیس IBookView، متد Show در BookShell فراخوانی خواهد شد که نتیجه آن نمایش فرم مورد نظر است.

### طراحی Bootstrapper

در پروژه‌های ماژولار Bootstrapper از ملزومات جدانشدنی این گونه پروژه هاست. برای این کار ابتدا یک WPF Application دیگر به نام Bootstrapper ایجاد نماید. سپس ارجاعی به اسمبلی‌های زیر را در آن قرار دهید:

«Controllers»

«Views»

«ViewModels»

«Shell»

«System.ComponentModel.Composition»

«نصب بسته WAF با استفاده از nuget»

حال یک کلاس به نام AppBootstrapper ایجاد نمایید و کدهای زیر را در آن کپی نمایید:

```

public class AppBootstrapper
{
    public CompositionContainer Container
    {
        get;
        private set;
    }

    public AggregateCatalog Catalog
    {
        get;
        private set;
    }

    public void Run()
    {
        Catalog = new AggregateCatalog();
        Catalog.Catalogs.Add(new AssemblyCatalog(Assembly.GetExecutingAssembly()));

        Catalog.Catalogs.Add(new AssemblyCatalog(String.Format("{0}\\{1}",
Environment.CurrentDirectory, "Shell.dll")));
        Catalog.Catalogs.Add(new AssemblyCatalog(String.Format("{0}\\{1}",
Environment.CurrentDirectory, "ViewModels.dll")));
        Catalog.Catalogs.Add(new AssemblyCatalog(String.Format("{0}\\{1}",
Environment.CurrentDirectory, "Controllers.dll")));

        Container = new CompositionContainer(Catalog);

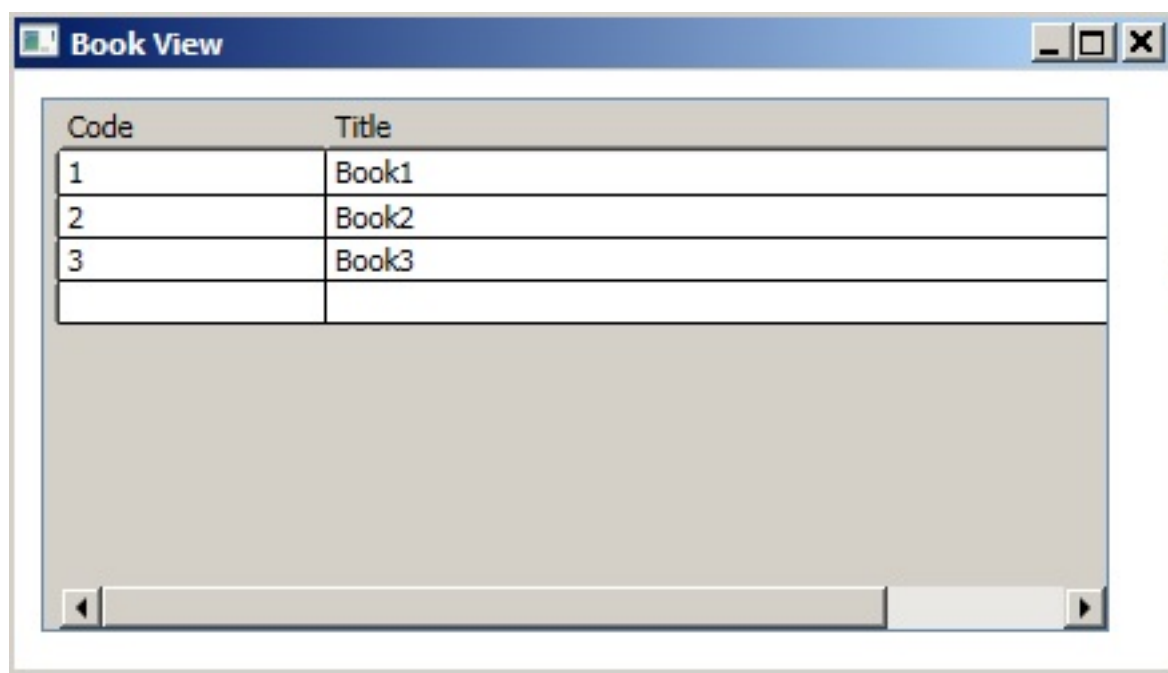
        var batch = new CompositionBatch();
        batch.AddExportedValue(Container);
        Container.Compose(batch);

        var bookController = Container.GetExportedValue<BookController>();
    }
}

```

```
        bookController.Run();  
    }  
}
```

اگر با MEF آشنا باشید کدهای بالا نیز برای شما مفهوم مشخصی دارند. در متد Run این کلاس ابتدا Catalog ساخته می‌شود. سپس با اسکن اسمبلی‌های مورد نظر تمام Exportها و Importهای لازم واکنشی شده و به Conrtainer مورد نظر رجیستر می‌شوند. در انتها نیز با وهله سازی از BookController و فراخوانی متد Run آن خروجی زیر نمایان خواهد شد.



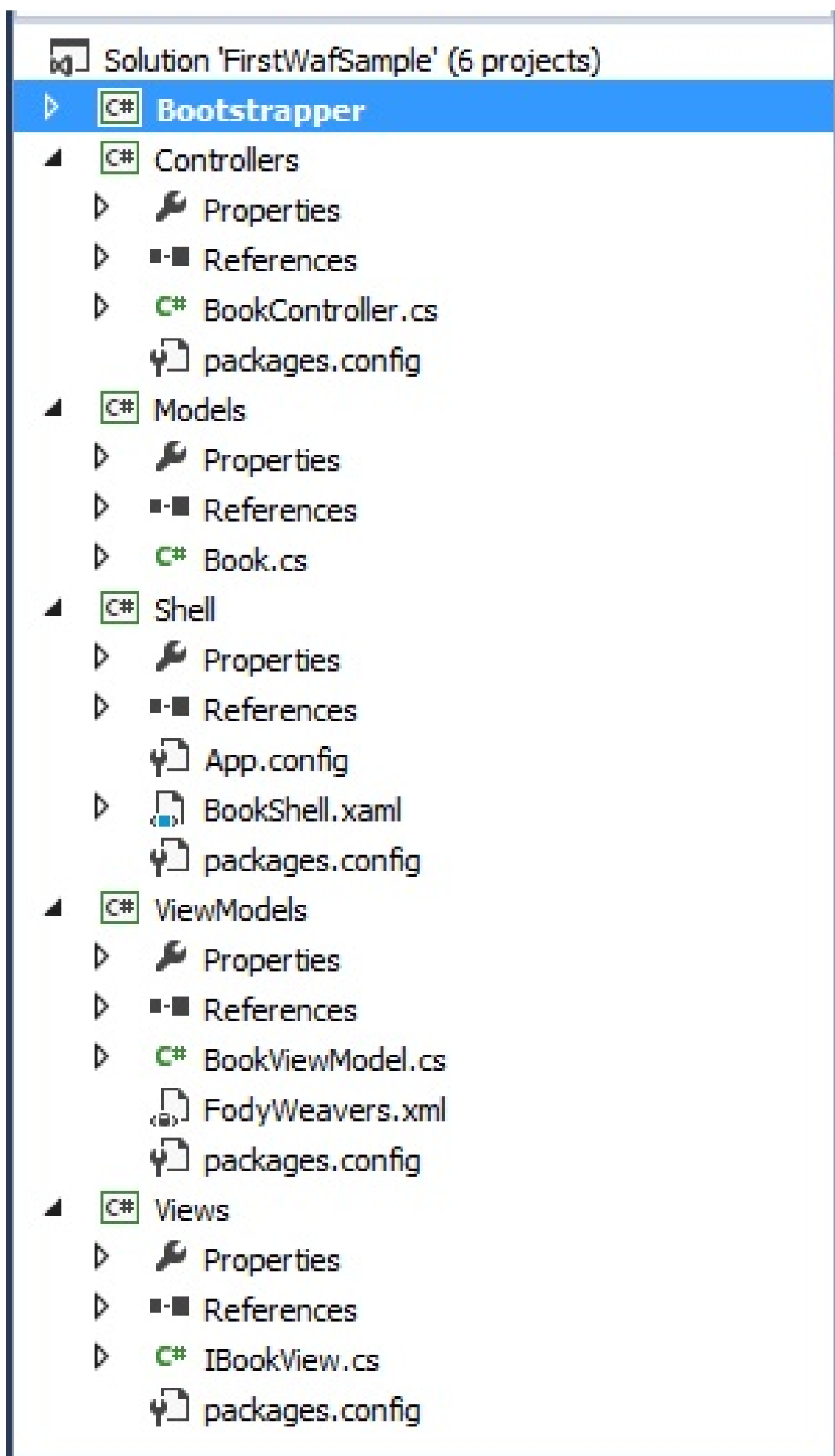
Code	Title
1	Book1
2	Book2
3	Book3

نکته بخش Startup را از فایل App.Xaml حذف نمایید و در متد Startup این فایل کد زیر را کپی کنید:

```
public partial class App : Application  
{  
    protected override void OnStartup(StartupEventArgs e)  
    {  
        new Bootstrapper.AppBootstrapper().Run();  
    }  
}
```

در پایان، ساختار پروژه به صورت زیر خواهد شد:





**نکته:** می‌توان بخش اسکن اسمبلی‌ها را توسط یک DirecotryCatalog به صورت زیر خلاصه کرد:

```
Catalog.Catalogs.Add(new DirectoryCatalog(Environment.CurrentDirectory));
```

در این صورت تمام اسمبلی‌های موجود در این مسیر اسکن خواهند شد.

**نکته:** می‌توان به جای جداسازی فیزیکی لایه‌ها آن‌ها را از طریق Directoryها به صورت منطقی در قالب یک اسمبلی نیز مدیریت کرد.

**نکته:** بهتر است به جای رفرنس مستقیم اسمبلی‌ها به Bootstrapper با استفاده از Pre post build در قسمت Build Event، اسمبلی‌های مورد نظر را در یک مسیر Build کپی نمایید که روش آن به تفصیل در این [پست](#) و این [پست](#) شرح داده شده است.

[دانلود سورس پروژه](#)

در این پست قصد داریم مثال [قسمت](#) قبل را توسعه داده و پیاده سازی Commandها را در آن در طی یک مثال بررسی کنیم. از این جهت دکمه‌ای، جهت حذف آیتم انتخاب شده در دیتا گرید، به فرم BookShell اضافه می‌نماییم. به صورت زیر:

```
<Button Content="RemoveItem" Command="{Binding RemoveItemCommand}" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="75"/>
```

Command تعریف شده در Button مورد نظر به خاصیتی به نام RemoveItemCommand در BookViewModel که نوع آن ICommand است اشاره می‌کند. پس باید تغییرات زیر را در ViewModel اعمال کنیم:

```
public ICommand RemoveItemCommand { get; set; }
```

از طرفی نیاز به خاصیتی داریم که به آیتم جاری در دیتاگرید اشاره کند.

```
public Book CurrentItem
{
    get
    {
        return currentItem;
    }
    set
    {
        if(currentItem != value)
        {
            currentItem = value;
            RaisePropertyChanged("CurrentItem");
        }
    }
}
private Book currentItem;
```

همان طور که در پست قبلی توضیح داده شد پیاده سازی‌ها تعاریف در ViewModel در Controller انجام می‌گیرد برای همین منظور باید تعریف DelegateCommand که یک پیاده سازی خاص از ICommand است در کنترلر انجام شود. :

```
[Export]
public class BookController
{
    [ImportingConstructor]
    public BookController(BookViewModel viewModel)
    {
        ViewModelCore = viewModel;
    }

    public BookViewModel ViewModelCore
    {
        get;
        private set;
    }

    public DelegateCommand RemoveItemCommand
    {
        get;
        private set;
    }

    private void ExecuteRemoveItemCommand()
    {
        ViewModelCore.Books.Remove(ViewModelCore.CurrentItem);
    }

    private void Initialize()
    {
        RemoveItemCommand = new DelegateCommand(ExecuteRemoveItemCommand);
        ViewModelCore.RemoveItemCommand = RemoveItemCommand;
    }
}
```

```

    }

    public void Run()
    {
        var result = new List<Book>();
        result.Add(new Book { Code = 1, Title = "Book1" });
        result.Add(new Book { Code = 2, Title = "Book2" });
        result.Add(new Book { Code = 3, Title = "Book3" });

        Initialize();

        ViewModelCore.Books = new ObservableCollection<Models.Book>(result);

        (ViewModelCore.View as IBookView).Show();
    }
}

```

### تغییرات:

«خاصیتی به نام RemoveItemCommand که از نوع DelegateCommand است تعریف شده است؛  
 «متدی به نام Initialize اضافه شد که متدهای Execute و CanExecute برای Command ها را در این قسمت رجیستر می‌کنیم.  
 «در نهایت Command تعریف شده در کنترلر به Command مربوطه در ViewModel انتساب داده شد.

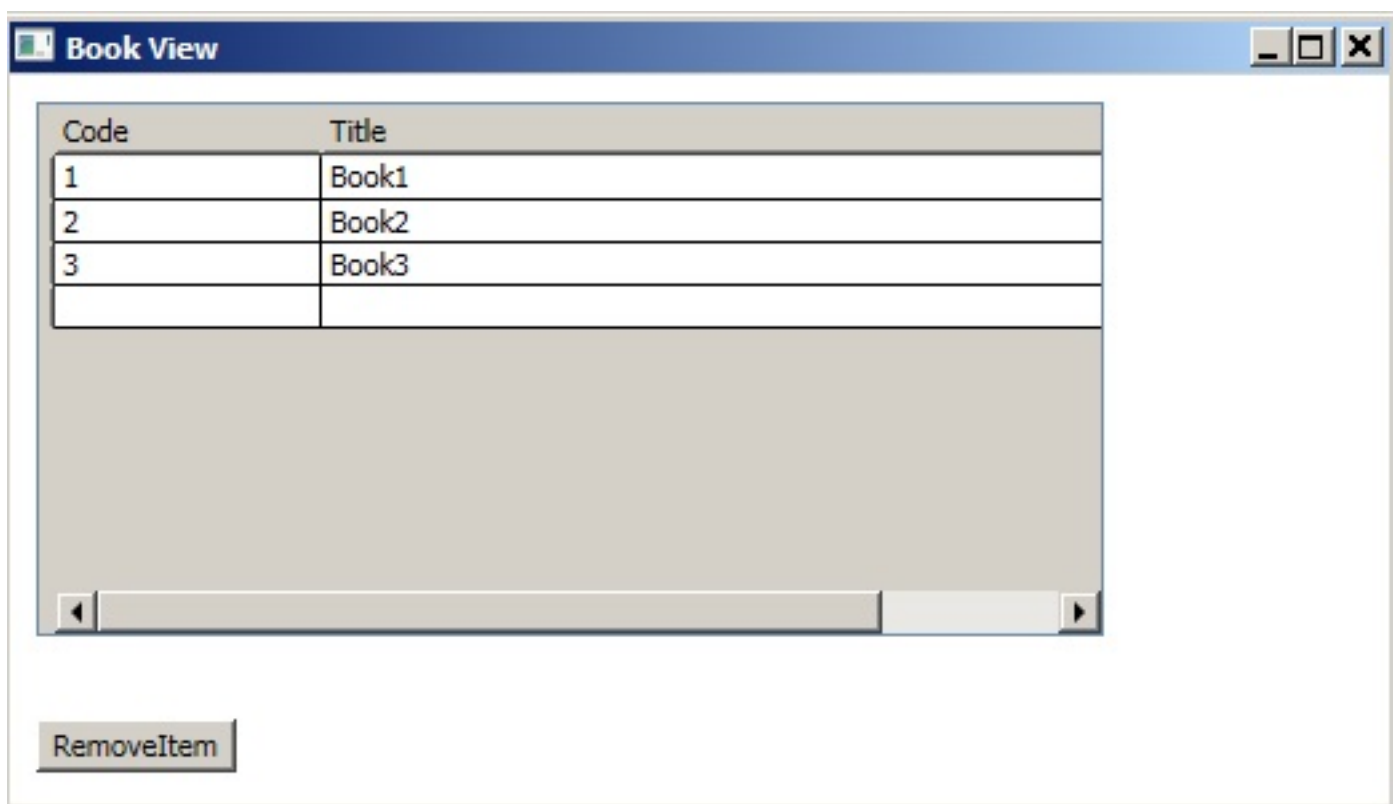
حال کافیت خاصیت SelectedItem دیتاگرید BookShell به خاصیت CurrentItem موجود در ViewModel مقید شود:

```

<DataGrid ItemsSource="{Binding Books}" SelectedItem="{Binding CurrentItem
,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}" HorizontalAlignment="Left" Margin="10,10,0,0"
VerticalAlignment="Top" Width="400" Height="200">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Code" Binding="{Binding Code}"
Width="100"></DataGridTextColumn>
        <DataGridTextColumn Header="Title" Binding="{Binding Title}"
Width="300"></DataGridTextColumn>
    </DataGrid.Columns>
</DataGrid>

```

اگر پروژه را اجرا نمایید، بعد از انتخاب سطر مورد نظر و کلیک بر روی دکمه RemoveItem مورد زیر قابل مشاهده است:



1 reference

```
private void ExecuteRemoveItemCommand()  
{  
    ViewModelCore.Books.Remove(ViewModelCore.CurrentItem);  
}
```

قصد داریم در مثال [پست قبلی](#) برای Command مورد نظر، عملیات اعتبارسنجی را فعال کنیم. اگر با الگوی MVVM آشنایی داشته باشید می‌دانید که می‌توان برای Command ها اکشنی به عنوان CanExecute تعریف کرد و در آن عملیات اعتبارسنجی را انجام داد. اما از آن جا که پیاده سازی این روش زمانی مسیر است که تغییرات خواص ViewModel در دسترس باشد در نتیجه در WAF مکانیزمی جهت ردیابی تغییرات خواص ViewModel در کنترلر فراهم شده است. در نسخه‌های قبلی WAF (قبل از نسخه 3) هر کنترلر از کلاس پایه ای به نام Controller ارث می‌برد که متد هایی جهت ردیابی تغییرات در آن در نظر گرفته شده بود به صورت زیر:

```
public class MyController : Controller
{
    [ImportingConstructor]
    public MyController(MyViewModel viewModel)
    {
        ViewModelCore = viewModel;
    }

    public MyViewModel ViewModelCore
    {
        get;
        private set;
    }

    public void Run()
    {
        AddWeakEventListener(ViewModelCore , ViewModelCoreChanged)
    }

    private void ViewModelCoreChanged(object sender , PropertyChangedEventArgs e)
    {
        if(e.PropertyName=="CurrentItem")
        {
        }
    }
}
```

همان طور که مشاهده می‌کنید با استفاده از متد AddWeakEventListener توانستیم تمامی تغییرات خواص ViewModel مورد نظر را از طریق متد ViewModelCoreChanged ردیابی کنیم. این متد بر مبنای الگوی [WeakEvent](#) پیاده سازی شده است. البته این تغییرات فقط زمانی قابل ردیابی هستند که در ViewModel متد RaisePropertyChanged برای متد set خاصیت فراخوانی شده باشد.

از آنجا که در دات نت 4.5 یک پیاده سازی خاص از الگوی [WeakEvent](#) در کلاس PropertyChangedEventManager موجود در اسمبلی WindowsBase و فضای نام System.ComponentModel انجام شده است در نتیجه توسعه دهندگان این کتابخانه نیز تصمیم به استفاده از این روش گرفتند که نتیجه آن Obsolete شدن کلاس پایه کنترلر در نسخه‌های 3 به بعد آن است. در روش جدید کفایت به صورت زیر عمل نماید:

```
[Export]
public class BookController
{
    [ImportingConstructor]
    public BookController(BookViewModel viewModel)
    {
        ViewModelCore = viewModel;
    }

    public BookViewModel ViewModelCore
    {
        get;
        private set;
    }

    public DelegateCommand RemoveItemCommand
```

```

    {
        get;
        private set;
    }

    private void ExecuteRemoveItemCommand()
    {
        ViewModelCore.Books.Remove(ViewModelCore.CurrentItem);
    }

    private bool CanExecuteRemoveItemCommand()
    {
        return ViewModelCore.CurrentItem != null;
    }
    private void Initialize()
    {
        RemoveItemCommand = new DelegateCommand(ExecuteRemoveItemCommand ,
CanExecuteRemoveItemCommand);
        ViewModelCore.RemoveItemCommand = RemoveItemCommand;
    }

    public void Run()
    {
        var result = new List<Book>();
        result.Add(new Book { Code = 1, Title = "Book1" });
        result.Add(new Book { Code = 2, Title = "Book2" });
        result.Add(new Book { Code = 3, Title = "Book3" });

        Initialize();
        ViewModelCore.Books = new ObservableCollection<Models.Book>(result);

        PropertyChangedEventManager.AddHandler(ViewModelCore, ViewModelChanged, "CurrentItem");
        (ViewModelCore.View as IBookView).Show();
    }

    private void ViewModelChanged(object sender,PropertyChangedEventArgs e)
    {
        if(e.PropertyName == "CurrentItem")
        {
            RemoveItemCommand.RaiseCanExecuteChanged();
        }
    }
}

```

**تغییرات:**

«ابتدا متدی به نام CanExecuteRemoveItemCommand ایجاد کردیم و کدهای اعتبارسنجی را در آن قرار دادیم؛  
«هنگام تعریف Command مربوطه متد بالا را به DelegateCommand رجیستر کردیم:

```
RemoveItemCommand = new DelegateCommand(ExecuteRemoveItemCommand , CanExecuteRemoveItemCommand);
```

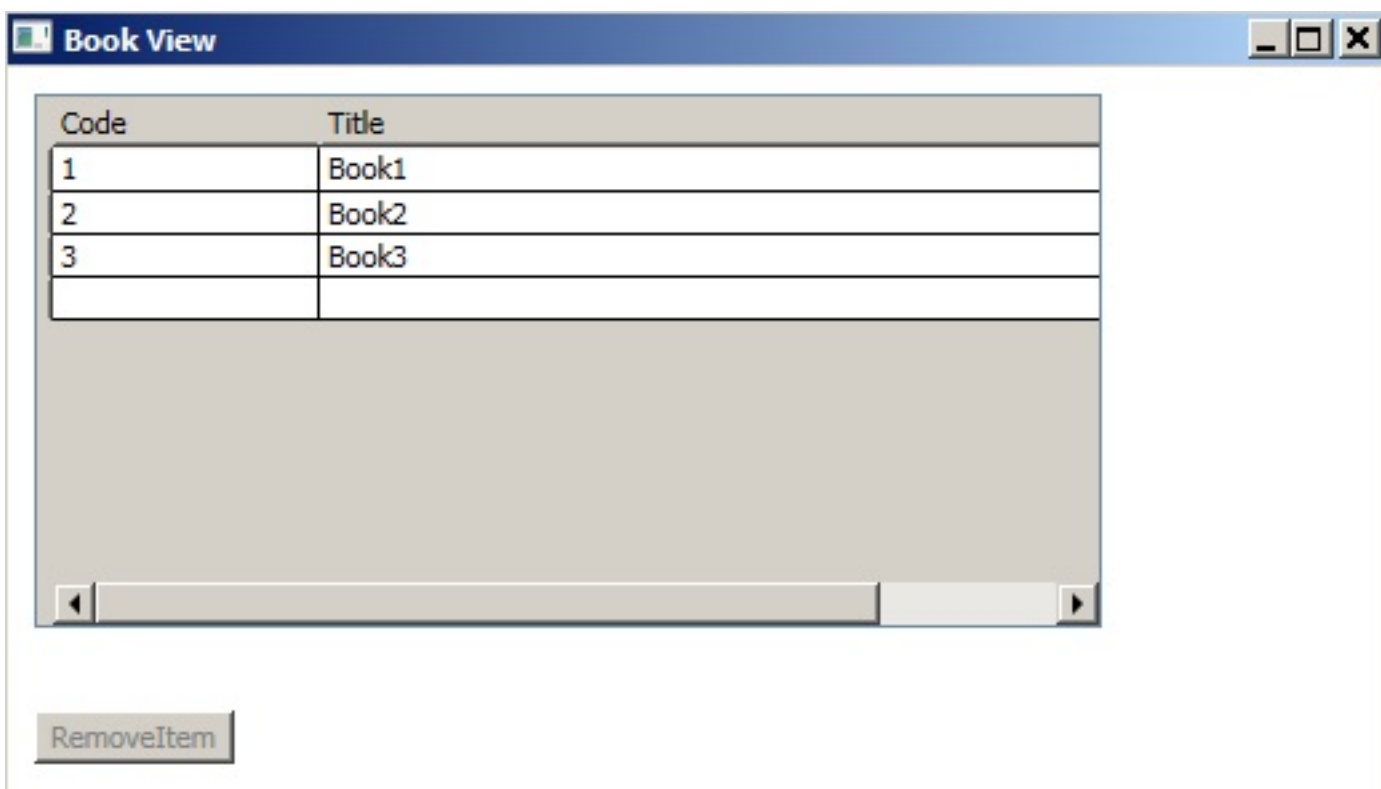
در این حالت بعد از اجرای برنامه همواره دکمه RemoveItem غیر فعال خواهد بود. دلیل آن این است که بعد از انتخاب آیتم مورد نظر از لیست باید کنترلر را متوجه تغییر در مقدار خاصیت CurrentItem نماییم. بدین منظور کد زیر را به متد Run اضافه کردم:

```
PropertyChangedEventManager.AddHandler(ViewModelCore, ViewModelChanged, "CurrentItem");
```

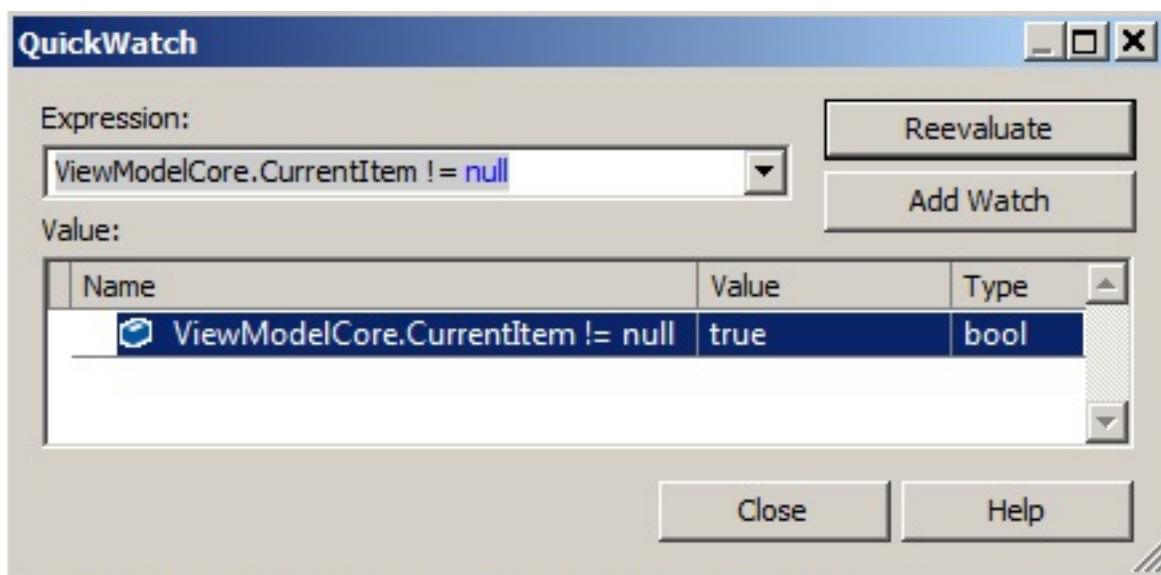
دستور بالا دقیقاً معادل دستور AddWeakEventListener موجود در نسخه‌های قدیمی WAF است. سپس در صورتی که نام خاصیت مورد نظر CurrentItem بود با استفاده از دستور RaiseCanExecuteChanged در کلاس DelegateCommand کنترلر را ملزم به اجرای دوباره متد CanExecuteRemoveItemCommand می‌کنیم.

اجرای برنامه:

ابتدا دکمه RemoveItem غیر فعال است:



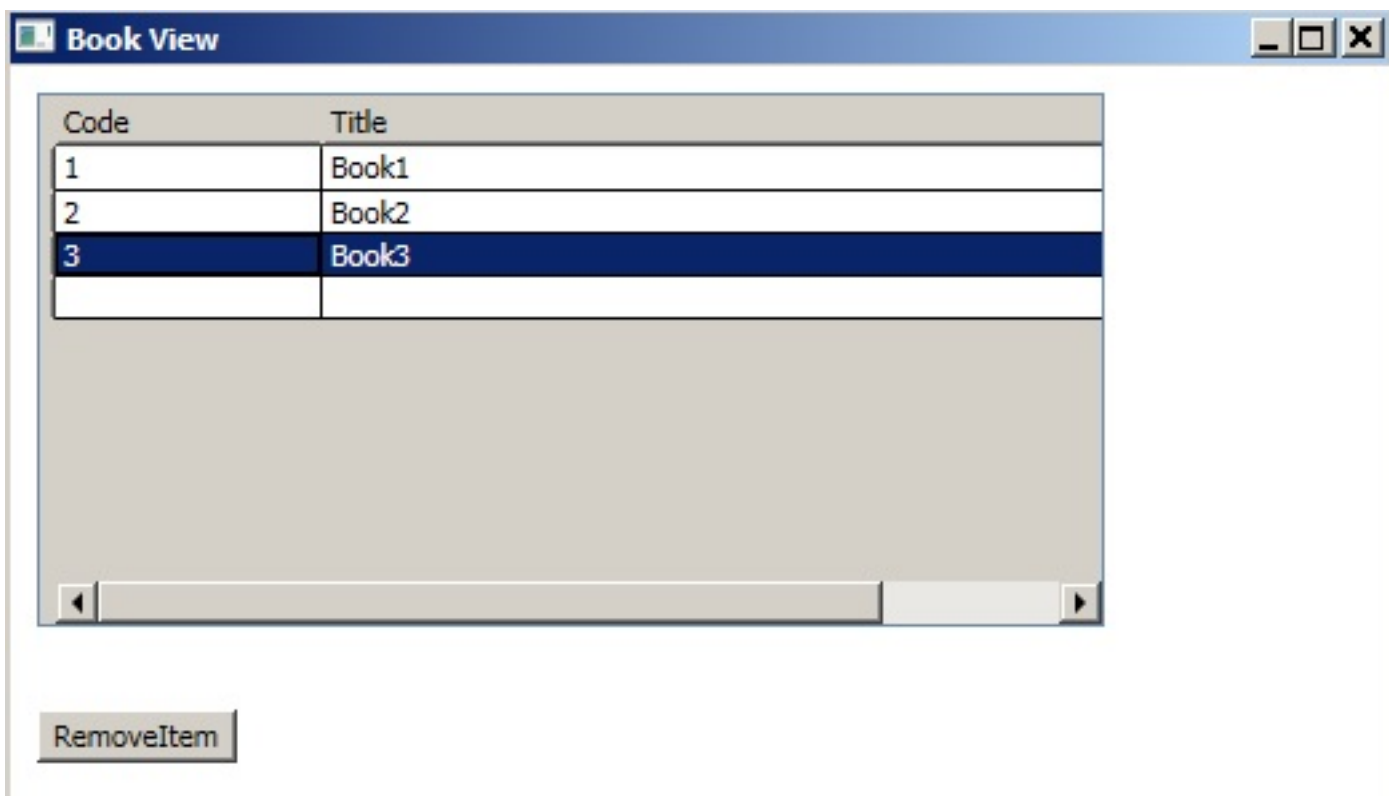
بعد از انتخاب یکی از گزینه و فراخوانی مجدد متد CanExecuteRemoveItemCommand دکمه مورد نظر فعال می‌شود:



```
private bool CanExecuteRemoveItemCommand()
{
    return ViewModelCore.CurrentItem != null;
}
```



و در نهایت دکمه RemoveItem فعال خواهد شد:



[دانلود سورس پروژه](#)