Introduction

In this article I’ll explain how to overcome several common challenges when creating unit testable WPF application using MVVM. I included the sample code of the demo application accompanying this article. The source code can be downloaded from GitHub.

It is a very simple WPF application that allows the user to work with person directory. The user can view person directory, view person details, edit person data, add a new person to directory or remove existing one. When deleting a person, the application displays a message box for confirming deletion.
Even though the application is simple, it has enough functionality to help me better explain how to overcome common challenges when writing testable WPF app.

**Dispatcher**

MVVM design pattern helps separate application logic from the user interface. However, UI updates need to happen on the UI thread. All changes made via `INotifyPropertyChanged` are automatically marshaled to the UI thread, but when updating collections that implement `INotifyCollectionChanged` (`ObservableCollection<T>`), the changes are not marshaled to the UI thread. If you try to modify an `ObservableCollection` which is bound to some WPF collection view on a thread other than UI thread, you will get `NotSupportedException` with a message: *This type of CollectionView does not support changes to its SourceCollection from a thread different from the Dispatcher thread.*

*If you are interested to know more about the Dispatcher, there is a great post by Shawn Wildermuth on MSDN.*

The obvious answer to the problem would be to wrap the code responsible for updating the collection in `Dispatcher.Invoke`.

```
personDirectory.AddRange(persons));
```

This would work, but now we ran into another problem. We are unable to test the code that uses `App.Current.Dispatcher` (since `App.Current` will be null during unit tests execution).

A possible solution would be to create an interface `IDispatcher` and a wrapper around `App.Current.Dispatcher` that implements that interface.
Then, a dependency to the **IDispatcher** interface should be introduced to the ViewModel. Now, we can use any of the Dependency Injection (**DI**) containers to provide a **DispatcherWrapper** at runtime.

To foster DRY principle, it’s convenient to create a base class for all your ViewModels, and add common dependencies to it.
public abstract class ViewModelBase : NotificationObject
{
    protected IDispatcher dispatcher;

    public ViewModelBase(IDispatcher dispatcher)
    {
        if (dispatcher == null)
        {
            throw new ArgumentNullException("dispatcher");
        }

        this.dispatcher = dispatcher;
    }

    ...
}

/// <summary>
/// Interaction logic for App.xaml
/// </summary>
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        IUnityContainer container = new UnityContainer();
        container.RegisterInstance<IDispatcher>(new DispatcherWrapper());
    }

    For unit tests, you can mock IDispatcher interface or create a fake implementation.
Modal dialogs

One of the most common tasks in MVVM is displaying a message box from the ViewModel object, whether the reason is to display some important notification to the user, or to ask for confirmation. This looks fairly simple. Calling `MessageBox.Show()` might do the job. There are couple of problems with this approach. First, it won’t work if we would like to show a custom made message box (or a dialog) and second, the message box will be presented during execution of unit tests. A good unit test is able to be fully automated. By displaying the message box, we prevent the test suite to complete. We need to find a way to somehow display message boxes (or dialogs) at runtime and simulate user response while running unit tests.

The solution to the problem is to create an interface `IDialogService` and the concrete service that implements the interface and wraps `Window.ShowDialog()` and `MessageBox.Show()` methods. I used a sample from the CodeProject for my demo app.

After introducing IDialogService as dependency to our ViewModelBase class and registering the DialogService with the DI container, we display the message box at runtime, and we can mock the service in unit test environment.
private void DeletePerson()
{
    if (dialogService.ShowMessageBox(this, "Are you sure?", "Confirm deletion", MessageBoxButton.YesNo, MessageBoxIcon.Question) == MessageBoxResult.Yes)
    {
        if (personService.DeletePerson(this.person.Id))
        {
            ...
        }
    }
}

container.RegisterInstance<IDialogService>(new DialogService());

Now, with Moq, we can mock the DialogService behavior, but not just that, we can as well test whether or not ShowMessageBox was called and even if it was called with the correct arguments.

protected Mock<IDialogService> dialogServiceMock = new Mock<IDialogService>();

/// <summary>
/// Simulates user answer to presented confirmation dialog
/// </summary>
private void WhenDeleteConfirmationIsReceived()
{
    dialogServiceMock.Setup(x => x.ShowMessageBox(
        It.IsAny<object>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<MessageBoxImage>(),
        It.IsAny<MessageBoxButton>())).Returns(MessageBoxResult.Yes);
}

protected void AssertMessageBoxWasDisplayed(string failMessage)
{
    dialogServiceMock.Verify(x => x.ShowMessageBox(
        It.IsAny<object>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<MessageBoxImage>(),
        It.IsAny<MessageBoxButton>())).Returns(MessageBoxResult.Yes);
}
protected void AssertValidMessageBoxWasDisplayed(object expectedViewModel,
string expectedText, string expectedCaption, MessageBoxButton expectedButtons,
MessageBoxImage expectedImage, string failMessage)
{
    dialogServiceMock.Verify(x => x.ShowMessageBox(
        It.Is<object>(o => o.Equals(expectedViewModel)),
        It.Is<string>(s => s.Equals(expectedText)),
        It.Is<string>(s => s.Equals(expectedCaption)),
        It.Is<MessageBoxButton>(b => b.Equals(expectedButtons)),
        It.Is<MessageBoxImage>(i => i.Equals(expectedImage))),
    Times.Once, failMessage);
}

[TestMethod]
public void PersonDetailsViewModel_GivenPerson_WhenDeleteCommandIsExecuted_ThenUserShouldBePromptedToConfirm()
{
    GivenPerson();
    viewModel.DeletePersonCommand.Execute(null);
    AssertValidMessageBoxWasDisplayed(
        viewModel,
        "Are you sure?",
        "Confirm deletion",
        MessageBoxButton.YesNo,
Communication between ViewModels

When you develop composite UI with several parts that need to be synchronized with each other, you need to somehow connect those parts.

This can be done by publishing and subscribing to .NET events, or by tightly coupling a publisher with all its subscribers (directly calling subscriber class’ method from the publisher), thus making the application hard to maintain.

Event Aggregator Pattern helps decoupling publisher and subscribers. Publishers and subscribers don’t communicate directly, but instead they only have a reference to EventAggregator service, which acts as a container for events.

EventAggregator has publication and subscription methods which are used by publishers and subscribers respectively.

In my demo app, I have used Prism’s EventAggregator service which implements IEventAggregator interface.

```csharp
public class PersonDeletedEvent : CompositePresentationEvent<Person>
```
public class PersonDetailsViewModel : ViewModelBase
{
    private void DeletePerson()
    {
        ...
        if (personService.DeletePerson(this.person.Id))
        {
            aggregator.GetEvent<PersonDeletedEvent>().Publish(person);
        }
        ...
    }
}

public class PersonDirectoryViewModel : ViewModelBase
{
    readonly RangeEnabledObservableCollection<Person>
    personDirectory;
    Person selectedPerson;

    public PersonDirectoryViewModel(
        IPersonService personService,
        IDispatcher dispatcher,
        IEventAggregator aggregator,
        IDialogService dialogService)
        : base(personService, dispatcher, aggregator, 
        dialogService)
    {
        personDirectory =
            new RangeEnabledObservableCollection<Person>();

        aggregator.GetEvent<PersonDeletedEvent>().
            Subscribe(OnPersonDeleted,
private void OnPersonDeleted(Person person)
{
    this.personDirectory.Remove(person);
}

To provide unit test support it is required to mock both IEventAggregator and the Events it uses.

public class TestBase
{
    protected Mock<PersonDeletedEvent> personDeletedEventMock = new Mock<PersonDeletedEvent>();

    private void AggregatorSetup()
    {
        aggregatorMock.Setup(x => x.GetEvent<PersonDeletedEvent>())
        .Returns(personDeletedEventMock.Object);
    }
}

When testing whether the event was published with the correct payload, Moq Verify method is used on the event mock.

In this example, when DeletePersonCommand is executed, it is expected that PersonDeletedEvent was published.
public class PersonDetailsViewModelTest : TestBase
{
    PersonDetailsViewModel viewModel;
    private void WhenDeletionIsSuccessful()
    {
        ...
        viewModel.DeletePersonCommand.Execute(null);
    }
    ...

    [TestMethod]
    public void PersonDetailsViewModel_GivenPerson_WhenDeletionIsSuccessful_ThenPersonDeleteEventShouldBePublished()
    {
        GivenPerson();
        WhenDeletionIsSuccessful();

        personDeletedEventMock.Verify(x => x.Publish(
            It.Is<Person>(p => object.ReferenceEquals(p, viewModel.Person))),
            Times.Once, "Person deleted event was not published exactly once,
            or was published with wrong person as argument");
    }
}

When testing a subscriber, the challenge is to obtain a reference to a subscriber callback.
This can be done by setting up the event mock like in the following example.

After that, you just Invoke the method with a custom payload and do the assertions.
public class PersonDirectoryViewModelTest : TestBase
{
    
    Action<Person> personDeletedCallback;

    private void WhenPersonDeletedEventIsReceived(Person person)
    {
        personDeletedCallback.Invoke(person);
    }

    ...

    [TestMethod]
    public void PersonDirectoryViewModel_GivenPersonDirectory_WhenPersonDeletedEventIsReceived_PersonShouldBeRemovedFromDirectory()
    {
        personDeletedEventMock.Setup(  
            x =>  
            x.Subscribe(  
                It.IsAny<Action<Person>>(),  
                It.IsAny<ThreadOption>(),  
                It.IsAny<bool>(),  
                It.IsAny<Predicate<Person>>()))
            .Callback<Action<Person>, ThreadOption, bool, Predicate<Person>>(
            (e, t, b, a) => personDeletedCallback = e);
        
        GivenPersonDirectory();
        WhenPersonDeletedEventIsReceived(persons[0]);
        CollectionAssert.DoesNotContain(
            viewModel.PersonDirectory, persons[0]);
    }

    Example application
    The app has a main view and dialog view (PersonDialogView) for adding/editing person data. Main view is composed of two views, PersonDirectoryView on the left, and PersonDetailsView
on the right. These views are bound to the corresponding ViewModels:

PersonDirectoryViewModel, PersonDetailsViewModel and PersonDialogViewModel.

Each ViewModel inherits from ViewModelBase, which has dependencies to all the required interfaces:

- **IPersonService** – service for accessing Person repository (PersonService is the concrete implementation that connects to fake database)
- **IDispatcher** – service for handling the UI updates (DispatcherWrapper is the concrete implementation used)
- **IEventAggregator** – service that facilitates the communication between ViewModels (Prism’s EventAggregator is the concrete implementation used)
- **IDialogService** – service used for displaying dialogs and message boxes from ViewModels (DialogService is the concrete implementation used)

The ViewModelBase itself inherits from Microsoft.Practices.Prism.MvvmBindableBase, which is a base class for items that support property notification.

This class provides basic support for implementing INotifyPropertyChanged.

The ViewModelBase’s **IsBusy** property is used to make the UI more responsive. The long running operations, such as data loading can be done asynchronously, on a background thread, and the entire view (or its part) can be wrapped in BusyIndicator control, that is bound to IsBusy property. The control displays a waiting message, whenever the value of its IsBusy property is true. BusyIndicator control is part of Extended WPF Toolkit.
The Person model class represents a person with an id, first- and last name, an age and a full name (readonly).

Instances of this class are wrapped in ViewModels, whether as association, or as a part of the collection. The same instance can be shared between different ViewModels, and because the Person class also inherits from BindableBase, all the interested parties will be notified if any of the Person data changes.
This would suffice for updating person data through one view, and observing the changes in another. However, for the case of adding a new person, deleting the existing one, or changing currently selected person, the ViewModels communicate through EventAggregator by using these three events:

- **SelectedPersonChangeEvent** - published by PersonDirectoryViewModel when selected person is changed; received by PersonDetailsViewModel
- **PersonCreatedEvent** - published by PersonDetailsViewModel when new person is created; received by PersonDirectoryViewModel
- **PersonDeletedEvent** - published by PersonDetailsViewModel when a person is deleted; received by PersonDirectoryViewModel

These events inherit from `CompositePresentationEvent<TPayload>`, where TPayload is the type of message that will be sent to subscribers, in this case a Person class. **PersonDetailsViewModel** updates its **Person** property upon receiving **SelectedPersonChangeEvent** and raises **CanExecuteChanged** for **EditPersonCommand** and **DeletePersonCommand**. Upon receiving **PersonCreatedEvent/PersonDeletedEvent**, **PersonDirectoryViewModel** adds/removes a **Person** to/from **PersonDirectory**, which is an **ObservableCollection**, so a view bound to that collection is automatically updated.

**Conclusion**

In this post, I have used the sample application to illustrate how to write a testable MVVM application: wrapping and mocking the dispatcher, displaying modal dialog and testing if it
was called with correct arguments, using the EventAggregator pattern for communication between ViewModels and testing publisher and subscribers.