A Message Broker for XAML Applications

In any application, you want to keep the coupling between any two or more objects as loose as possible. Coupling happens when one class contains a property that is used in another class or uses another class in one of its methods. If you have this situation, then this is called strong or tight coupling. One popular design pattern to help with keeping objects loosely coupled is called the mediator design pattern. The basics of this pattern are very simple; avoid one object directly talking to another object, and instead use another class to mediate between the two. This class is called a message broker. The purpose of this blog post is show you a simple approach to using a message broker in your XAML applications.

Uses of the Message Broker

Consider how you talk to your friends in person versus how you text message them. If you talk to them in person, you must both agree on where and when to meet. This creates a dependency. Now, think about how you send a text message to one of your friends. Your phone has an application into which you type a message and your service provider is the mediator who is responsible for delivering that message. You rely on the mediator to broadcast that message, and you rely on their phone being on and checking for those messages.

WPF applications consist of many windows, user controls, view models, and other classes. It is very common that you need to change a property (send a message) in view model X from within a method in view model Y. Or, maybe you want to display a status message on the main window from a user control. One way to accomplish these tasks is to pass a reference of the current component to the other component when the new component is instantiated. However, this makes these components dependent on one another. This makes each component harder to reuse, test, and maintain.

Instead of creating a dependency from one window to another window, you can create a mediator class to which you send a message. This mediator class broadcasts that message, and another component can do something with that message if it is checking for it. For example, view model X can use the mediator class to send a message named "ChangeLastName", and the payload it is sending is "Smith". View model Y can be listening for the message named "ChangeLastName" and when it receives that message, can grab the payload

"Smith" and assign that value to its *LastName* property. View model X knows nothing about view model Y or what that view model might do with the message it sends. This means there is no dependency between view model X and view model Y. They each just have a dependency on the mediator (service provider) class that knows how to send and receive messages. Having just a single, simple, class that all components in your application use is much better than having multiple dependencies between all the various components in your application.

In Figure 1, you see an example of a main window (1) and a user control on that window (2). The user control (2) needs to display a message "Product List" (3) on the main window. Instead of referencing the main window, the user control sends a "DisplayStatusMsg" message through the mediator class passing in "Product List" as its payload. The main window responds to events sent through the mediator class and when it receives the "DisplayStatusMsg" message, it takes the payload and assigns it to the *Text* property of a TextBlock control.

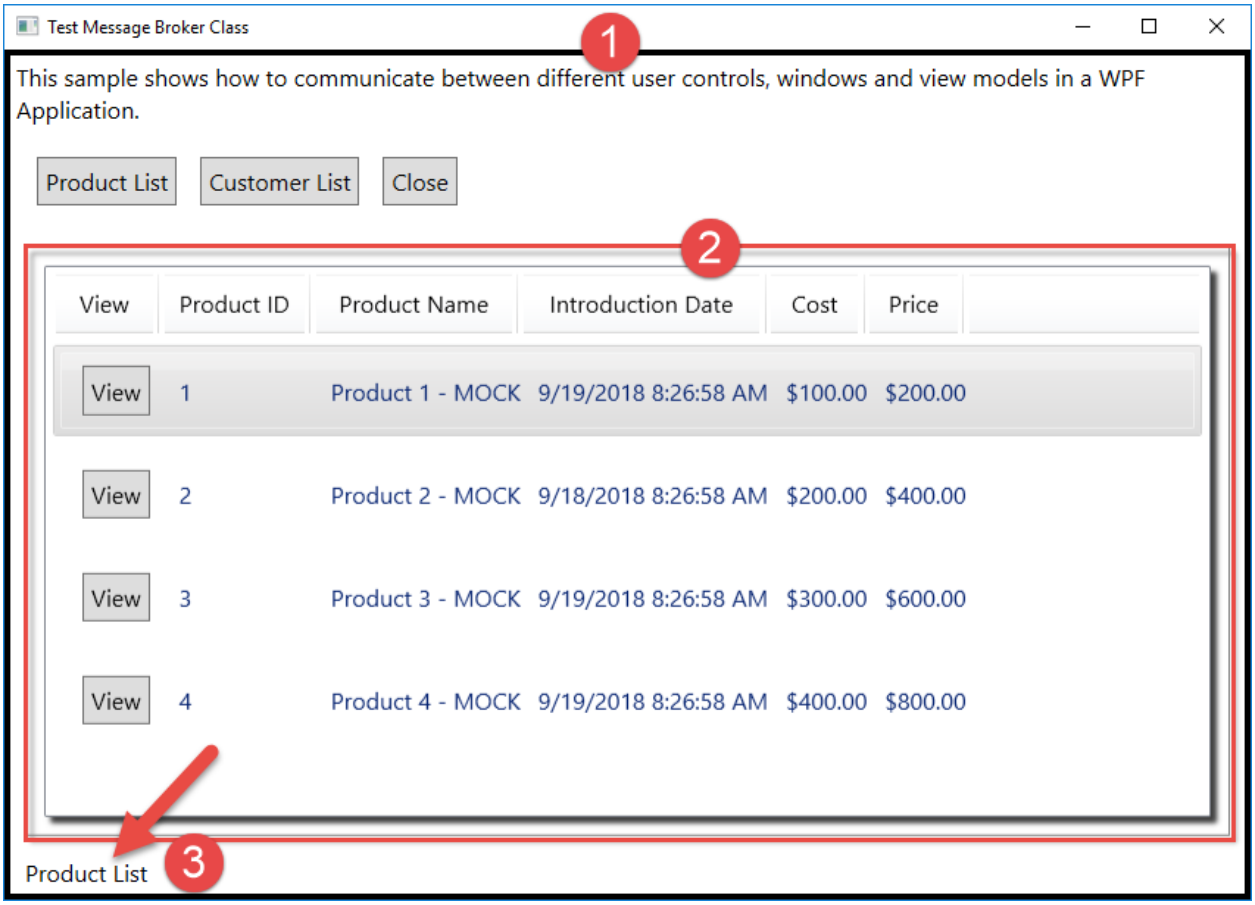

Figure 1: You can use a message broker to pass messages from a user control to a window.

The Message Broker Class

The main class that sends the messages and defines the event signature for receiving messages is called **MessageBroker**. This class has the following elements.

- A delegate for the message received event
- A singleton property
- A SendMessage method
- A MessageReceived event
- A RaiseMessageReceived method

The message broker sends a message via a normal .NET event. Two arguments are passed to an event—the object that raised the event and an instance of an **EventArgs** class. You are going to create a new class that inherits from **EventArgs** but adds two additional properties. One property is the name of the message being sent. It is this name that other objects can look for to see if they need to do something with this message. For example, a name might be "DisplayStatusMsg", "DisplayUserControl", "CloseWindow", etc. The second property is any payload you wish to send along with the message name. For example, if you send a "DisplayStatusMsg" message, the payload might be a string to display in a status bar on the main window of your application. If the message is "DisplayUserControl", the payload might be a user control object to display on the main window.

The source for the **MessageBroker** class is shown below.

```
public class MessageBroker
{
  public delegate void MessageReceivedEventHandler(object sender,
           MessageBrokerEventArgs e);

  public event MessageReceivedEventHandler MessageReceived;

  private static MessageBroker _Instance;
  public static MessageBroker Instance
  {
    get {
      if (_Instance == null) {
        _Instance = new MessageBroker();
      }

      return _Instance;
    }
    set { _Instance = value; }
  }

  public void SendMessage(string messageName)
  {
    SendMessage(messageName, null);
  }

  public void SendMessage(string messageName, object payload)
  {
    MessageBrokerEventArgs arg;

    arg = new MessageBrokerEventArgs(messageName, payload);

    RaiseMessageReceived(arg);
  }

  protected void RaiseMessageReceived(MessageBrokerEventArgs e)
  {
    if (null != MessageReceived) {
      MessageReceived(this, e);
    }
  }
}
```

To raise an event from your class, first define a delegate so consumers of your event know the signature of that event. Next, declare the event as a type of MessageReceivedEventHandler and assign a name for the event; in this case "MessageReceived."

Since this class is going to be used as the single point of communication among all components in your application, create a singleton for this class. The name of this singleton is called *Instance*.

The SendMessage method take two parameters: the message name and the payload. It creates an instance of a **MessageBrokerEventArgs** class, passing to the constructor the two parameters: *messageName* and *payload*.

MessageBrokerEventArgs Class

The **MessageBrokerEventArgs** class inherits from the **System.EventArgs** class and adds a couple of additional properties needed for our message broker system. The properties are *MessageName* and *MessagePayload*. The *MessageName* property is a string value. The *MessagePayload* property is an object type so that any kind of data may be passed as a message.

```
public class MessageBrokerEventArgs : EventArgs
{
  public MessageBrokerEventArgs() : base()
  {
  }

  public MessageBrokerEventArgs(string messageName,
                                object payload) : base()
  {
    MessageName = messageName;
    MessagePayload = payload;
  }

  public string MessageName { get; set; }
  public object MessagePayload { get; set; }
}
```

Sending and Receiving Messages

As shown in Figure 1, you have a main window (1) with a product list user control (2) added to it. When the user control is loaded, you should display a message "Product List" at the bottom of the main window. You send this type of message using the following syntax.

```
MessageBroker.Instance.SendMessage("DisplayStatusMsg",
                                   "Product List");
```

There is also a Close button on the main window. This button sends a message named "Close". The main window also checks for any "Close" messages passed. If that message is received, the main window removes the user control from itself. By having the main window respond to a "Close" message, you can send a close message from each user control via a button click on that user control.

```
MessageBroker.Instance.SendMessage("Close");
```

Receive a Message

For the above samples, you need to receive messages from the message broker class. In the constructor for the main window, add the following code to create an event procedure.

```
public MainWindow()
{
  InitializeComponent();

  // Initialize the Message Broker Events
  MessageBroker.Instance.MessageReceived +=
            Instance_MessageReceived;
}
```

When you add this line of code, an Instance_MessageReceived() event procedure is created. You now just need to check the *MessageName* property of the event argument to determine what to do with each message. The code below shows responding to "DisplayStatusMsg" and "Close" messages.

```
private void Instance_MessageReceived(object sender,
                                      MessageBrokerEventArgs e)
{
  switch (e.MessageName) {
    case "DisplayStatusMsg":
      statusBar.Text = e.MessageBody.ToString();
      break;
    case "Close":
      contentArea.Children.Clear();
      break;
  }
}
```

Use an Application Messages Class

Don't use hard-coded strings all over your application to send and receive messages. If you mistype one of them, it can be a hard bug to track down. Instead, create an **ApplicationMessages** class into which you place public static constants for each message you wish to send. Below is an example of such a message class.

```
public partial class ApplicationMessages
{
  public const string CUSTOMER_DETAIL_CHANGED =
    "CustomerDetailChanged";

  public const string DISPLAY_STATUS_MESSAGE =
    "DisplayStatusMessage";

  public const string CLOSE_USER_CONTROL =
    "CloseUserControl";
}
```

Remove Event Handler

As with most “global” classes or classes that hook up events to other classes, garbage collection is something you need to consider. Just the simple act of hooking up an event procedure to a global event handler creates a reference between your user control and the message broker in the App class. This means that even when your user control is removed from your UI, the class will still be in memory because of the reference to the message broker. It is up to you to make sure you remove those event handlers. If you don’t, the garbage collector cannot release those objects.

In each of your user controls, make sure to remove the event handler in the Unloaded event.

```
private void UserControl_Unloaded(object sender, RoutedEventArgs e)
{
  if (_MessageBroker != null) {
      _MessageBroker.MessageReceived -=
        _MessageBroker_MessageReceived;
  }
}
```

Messages in a View Model Class

You can send and receive messages within a view model class just as you can from a user control or window. If you receive messages from within a view model class, you need to somehow remove that event handler. I would suggest creating a Dispose() method to remove it. Most .NET developers know to check for, and call a Dispose() method if one exists. You can go all the way and implement the **IDispose** interface, but it is unnecessary as we are not cleaning up any unmanaged resources.

The code below shows connecting to the MessageReceived event in the constructor of the **CustomerDetailViewModel** class. It also illustrates the creation of the Dispose() method to remove that event.

```
public class CustomerDetailViewModel : ViewModelBase
{
  public CustomerDetailViewModel() : base()
  {
    MessageBroker.Instance.MessageReceived +=
       Instance_MessageReceived;
  }

  private void Instance_MessageReceived(object sender,
                          MessageBrokerEventArgs e)
  {
    switch (e.MessageName) {
      case ApplicationMessages.CUSTOMER_DETAIL_CHANGED:
        SelectedItem = ((Customer)e.MessagePayload);
        break;
    }
  }

  // OTHER PROPERTIES/METHODS HERE

  public void Dispose()
  {
    MessageBroker.Instance.MessageReceived -=
       Instance_MessageReceived;
  }
}
```

If this view model is used on a window or a user control, be sure to call the Dispose() method in the Unloaded event on that window or user control. In the code below, you see the instance of the CustomerDetailViewModel is retrieved from the UserControl.Resources section on the user control. In the Unloaded event on this user control, invoke the Dispose() method to remove the event handler.

```
public partial class ucCustomerDetail : UserControl
{
  public CustomerDetailViewModel ViewModel { get; set; }

  public ucCustomerDetail()
  {
    InitializeComponent();

    ViewModel =
      (CustomerDetailViewModel)this.Resources["viewModel"];
  }

  private void UserControl_Unloaded(object sender,
    System.Windows.RoutedEventArgs e)
  {
    ViewModel.Dispose();
  }
}
```

Summary

In this blog post, you learned how to create a simple message broker system that allows you to send messages from one component to another without having to create dependencies between components. This reduces the coupling between objects in your application. You do need to remember to get rid of any event handlers prior to your components going out of scope or you run the risk of having memory leaks and events being called even though you can no longer access the object that is responding to that event.

One downfall of using a message broker (or any mediator design pattern) is that following the logic of the application is more difficult. After all, you may not know exactly from where the message is coming from. However, with some good comments you can hopefully mitigate this problem.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog,” then select “A Message Broker for XAML Applications” from the dropdown list.