A Communication System 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. As with most of my blog posts, the purpose is to introduce you to a simple approach to using a message broker, not all of the fine details.

IPDSAMessageBroker Interface

As with most implementations of a design pattern, you typically start with an interface or an abstract base class. In this particular instance, an Interface will work just fine. The interface for our Message Broker class just contains a single method “SendMessage” and one event “MessageReceived”.

```
public delegate void MessageReceivedEventHandler(
 object sender, PDSAMessageBrokerEventArgs e);

public interface IPDSAMessageBroker
{
  void SendMessage(PDSAMessageBrokerMessage msg);

  event MessageReceivedEventHandler MessageReceived;
}
```

PDSAMessageBrokerMessage Class

As you can see in the interface, the SendMessage method requires a type of PDSAMessageBrokerMessage to be passed to it. This class simply has a MessageName which is a ‘string’ type and a MessageBody property which is of the type ‘object’ so you can pass whatever you want in the body. You might pass a string in the body, or a complete Customer object. The MessageName property will help the receiver of the message know what is in the MessageBody property.

```
public class PDSAMessageBrokerMessage
{
  public PDSAMessageBrokerMessage()
  {
  }

  public PDSAMessageBrokerMessage(string name, object body)
  {
    MessageName = name;
    MessageBody = body;
  }

  public string MessageName { get; set; }

  public object MessageBody { get; set; }
}
```

PDSAMessageBrokerEventArgs Class

As our message broker class will be raising an event that others can respond to, it is a good idea to create your own event argument class. This class will inherit from the System.EventArgs class and add a couple of additional properties. The properties are the **MessageName** and **Message**. The **MessageName** property is simply a string value. The **Message** property is a type of a PDSAMessageBrokerMessage class.

```
public class PDSAMessageBrokerEventArgs : EventArgs
{
  public PDSAMessageBrokerEventArgs()
  {
  }

  public PDSAMessageBrokerEventArgs(string name,
    PDSAMessageBrokerMessage msg)
  {
    MessageName = name;
    Message = msg;
  }

  public string MessageName { get; set; }

  public PDSAMessageBrokerMessage Message { get; set; }
}
```

PDSAMessageBroker Class

Now that you have an interface class and a class to pass a message through an event, it is time to create your actual PDSAMessageBroker class. This class implements the SendMessage method and will also create the event handler for the delegate created in your Interface.

```
public class PDSAMessageBroker : IPDSAMessageBroker
{
  public void SendMessage(PDSAMessageBrokerMessage msg)
  {
    PDSAMessageBrokerEventArgs args;

    args = new PDSAMessageBrokerEventArgs(
      msg.MessageName, msg);

    RaiseMessageReceived(args);
  }

  public event MessageReceivedEventHandler MessageReceived;

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

The SendMessage method will take a PDSAMessageBrokerMessage object as an argument. It then creates an instance of a PDSAMessageBrokerEventArgs class, passing to the constructor two items: the MessageName from the PDSAMessageBrokerMessage object and also the object itself. It may seem a little redundant to pass in the message name when that same message name is part of the message, but it does make consuming the event and checking for the message name a little cleaner – as you will see in the next section.

Create a Global Message Broker

In your WPF application, create an instance of this message broker class in the App class located in the App.xaml file. Create a public property in the App class and create a new instance of that class in the OnStartUp event procedure as shown in the following code:

```
public partial class App : Application
{
  public PDSAMessageBroker MessageBroker { get; set; }

  protected override void OnStartup(StartupEventArgs e)
  {
    base.OnStartup(e);

    MessageBroker = new PDSAMessageBroker();
  }
}
```

Sending and Receiving Messages

Let's assume you have a user control that you load into a control on your main window and you want to send a message from that user control to the main window. You might have the main window display a message box, or put a string into a status bar as shown in Figure 1.

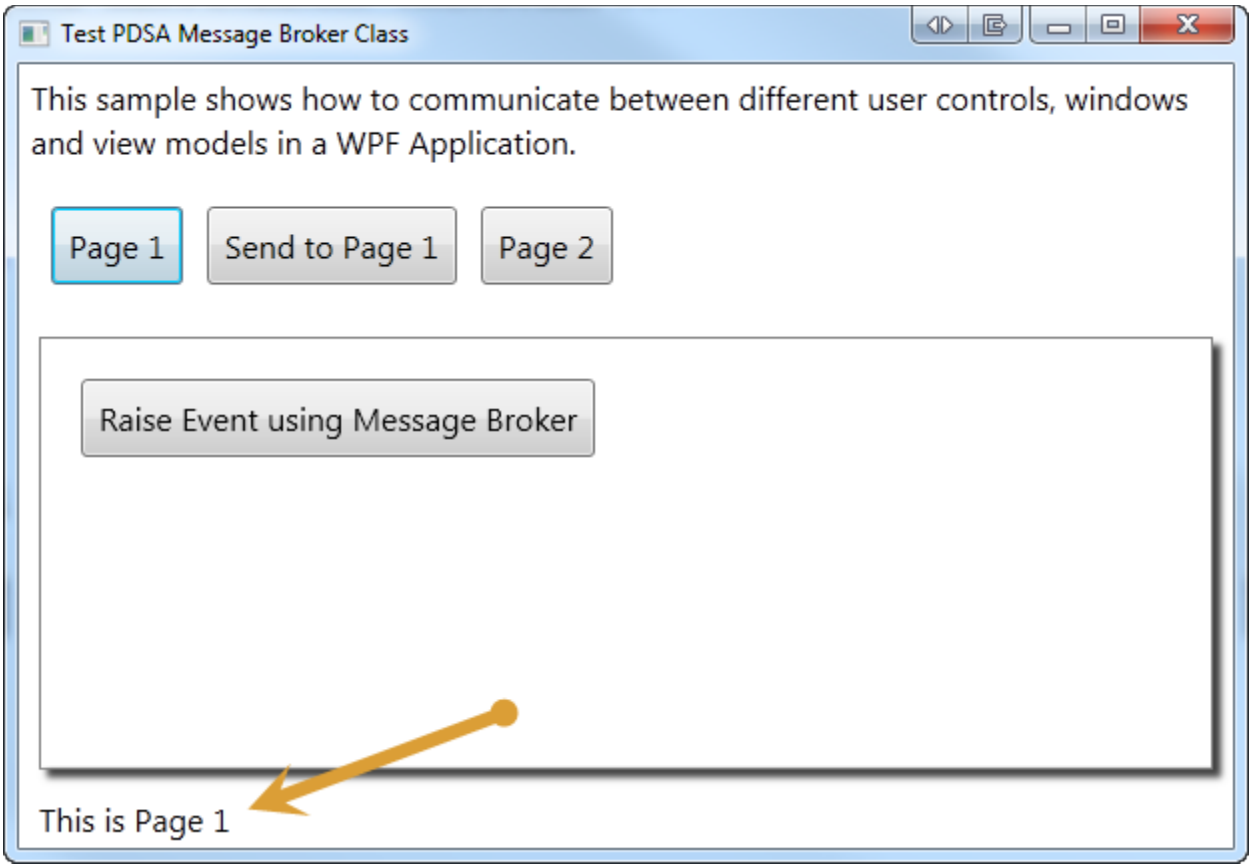

Figure 1: The main window can receive and send messages

The first thing you do in the main window is to hook up an event procedure to the MessageReceived event of the global message broker. This is done in the constructor of the main window:

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

  (Application.Current as App).MessageBroker.
     MessageReceived += new MessageReceivedEventHandler(
       MessageBroker_MessageReceived);
}
```

One piece of code you might not be familiar with is accessing a property defined in the App class of your XAML application. Within the App.Xaml file is a class named App that inherits from the Application object. You access the global instance of this App class by using **Application.Current**. You cast Application.Current to 'App' prior to accessing any of the public properties or methods you defined in the App class. Thus, the code **(Application.Current as App).MessageBroker**, allows you to get at the MessageBroker property defined in the App class.

In the MessageReceived event procedure in the main window (shown below) you can now check to see if the MessageName property of the PDSAMessageBrokerEventArgs is equal to “StatusBar” and if it is, then display the message body into the status bar text block control.

```
void MessageBroker_MessageReceived(object sender,
   PDSAMessageBrokerEventArgs e)
{
  switch (e.MessageName)
  {
    case "StatusBar":
      tbStatus.Text = e.Message.MessageBody.ToString();
      break;
  }
}
```

In the Page 1 user control’s Loaded event procedure you will send the message “StatusBar” through the global message broker to any listener using the following code:

```
private void UserControl_Loaded(object sender,
 RoutedEventArgs e)
{
  // Send Status Message
  (Application.Current as App).MessageBroker.
    SendMessage(new PDSAMessageBrokerMessage("StatusBar",
      "This is Page 1"));
}
```

Since the main window is listening for the message ‘StatusBar’, it will display the value “This is Page 1” in the status bar at the bottom of the main window.

Sending a Message to a User Control

The previous example sent a message from the user control to the main window. You can also send messages from the main window to any listener as well. Remember that the global message broker is really just a broadcaster to anyone who has hooked into the MessageReceived event.

In the constructor of the user control named ucPage1 you can hook into the global message broker’s MessageReceived event. You can then listen for any messages that are sent to this control by using a similar switch-case structure like that in the main window.

```
public ucPage1()
{
  InitializeComponent();

  // Hook to the Global Message Broker
  (Application.Current as App).MessageBroker.
    MessageReceived += new MessageReceivedEventHandler(
      MessageBroker_MessageReceived);
}

void MessageBroker_MessageReceived(object sender,
 PDSAMessageBrokerEventArgs e)
{
  // Look for messages intended for Page 1
  switch (e.MessageName)
  {
    case "ForPage1":
      MessageBox.Show(e.Message.MessageBody.ToString());
      break;
  }
}
```

Once the ucPage1 user control has been loaded into the main window you can then send a message using the following code:

```
private void btnSendToPage1_Click(object sender,
 RoutedEventArgs e)
{
  PDSAMessageBrokerMessage arg =
    new PDSAMessageBrokerMessage();

  arg.MessageName = "ForPage1";
  arg.MessageBody = "Message For Page 1";

  // Send a message to Page 1
  (Application.Current as App).MessageBroker.SendMessage(arg);
}
```

Since the MessageName matches what is in the ucPage1 MessageReceived event procedure, ucPage1 can do anything in response to that event. It is important to note that when the message gets sent it is sent to all MessageReceived event procedures, not just the one that is looking for a message called “ForPage1”. If the user control ucPage1 is not loaded and this message is broadcast, but no other code is listening for it, then it is simply ignored.

Remove Event Handler

In each class where you add an event handler to the MessageReceived event you need to make sure to remove those event handlers when you are done. Failure to do so can cause a strong reference to the class and thus not allow that object to be garbage collected. In each of your user control's make sure in the Unloaded event to remove the event handler.

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

Problems with Message Brokering

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. This can cause messages to still being handled even though the UI is not being displayed. It is up to you to make sure you remove those event handlers as discussed in the previous section. If you don't, then the garbage collector cannot release those objects.

Instead of using events to send messages from one object to another you might consider registering your objects with a central message broker. This message broker now becomes a collection class into which you pass an object and what messages that object wishes to receive. You do end up with the same problem however. You have to un-register your objects; otherwise they still stay in memory. To alleviate this problem you can look into using the WeakReference class as a method to store your objects so they can be garbage collected if need be. Discussing Weak References is beyond the scope of this post, but you can look this up on the web.

Summary

In this blog post you learned how to create a simple message broker system that will allow you to send messages from one object to another without having to reference objects directly. This does reduce the coupling between objects in your application. You do need to remember to get rid of any event handlers prior to your objects 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.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Tips & Tricks”, then select “A Communication System for XAML Applications” from the drop down list.