WPF Client Applications and Windows Workflow 4 Together

* 1. By Ron Jacobs
  2. Windows Workflow in .NET 4 was designed from the beginning to support running workflows in a variety of environments. The runtime itself is hidden but exposed to you via different types designed to support different scenarios you can learn more about this from the article [Using WorkflowInvoker and WorkflowApplication](http://msdn.microsoft.com/en-us/library/dd560894.aspx) on MSDN.
  3. In this post I'll show you what I learned about using WPF and WF together. I started by listing the requirements for the client application
  + Follows MVVM architecture
  + Runs workflows on background thread in the client process
  + Supports notification of events from Workflow
  + Supports control of workflow from UI (Cancel)
  + High test coverage
  + Implemented in TDD style (Test First - Red / Green / Refactor)

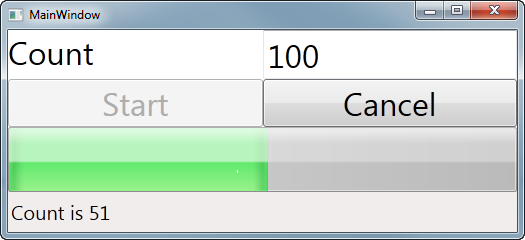
## Challenges

There are several challenges I needed to overcome to implement this solution.

* + How to use a Workflow in a WPF app built with MVVM architecture
  + How to support notifications from a workflow without messaging activities
  + How to control a workflow from the client app in MVVM

Now to show you how this sample addresses each of these challenges.

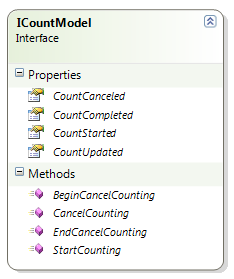
And what does this application do? It counts numbers. Yes, pointless I know but the example is just there to demonstrate a process (any process) controlled by a workflow from a WPF application.

* 1. 
  2. Figure 1
  3. The WPF Application Counts to 100 using a Workflow

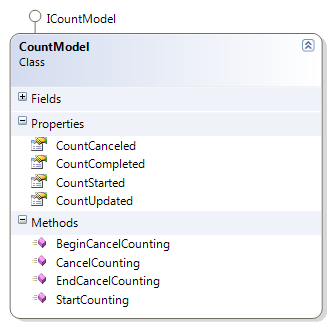
How to use a Workflow in a WPF app built with MVVM architecture

* 1. Workflows can run in any managed process. Many times they run as a service using IIS and Windows Server AppFabric. In this sample I want to run the workflow in the same process as the WPF app.
  2. In the MVVM architecture the logical place to do this is in the model. When I started building the application, I started with the model writing the test first and then implementing the model. Doing this helped me to get very high test coverage (nearly 90%) on the model.

### What is the purpose of the model?

* 1. The job of the model in MVVM is to represent the data and the processes that are exposed to the UI. The model knows nothing about the UI that it will be consumed by. This separation allows me to focus on creating an object that encapsulates the necessary business logic and data access code in a highly testable fashion.
  2. To start with I created the **ICountModel** interface to describe the capabilities I wanted in the model
  3. 
  4. Figure 2
  5. The Count Model
  6. Rather than use events, I created properties of Action<T> that allow the tests or View Model to receive notifications from the model about events in the workflow lifecycle.
  7. **Note:** I could have implemented these as events, I don't have a good reason for implementing them as Action<T> properties – I just thought I would try it.

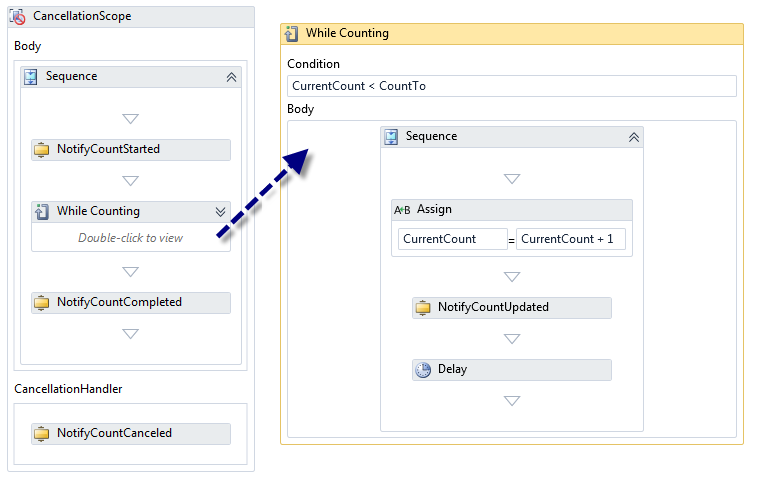
The **ICountModel** interface is implemented by the **internal** **CountModel** class. I made this class internal because I want the View Model to use ICountModel and not the CountModel class.

* 1. 
  2. Figure 3
  3. The CountModel class

How to support notifications from a workflow without messaging activities

* 1. Workflows can communicate with other apps and workflows using messaging activities. In my case I would be running the workflow within the host application's AppDomain so this allows me to take advantage of extensions.
  2. The WorkflowApplication.Extensions property contains a collection of Extensions which are simply objects (or functions that create objects) that you stuff in there. This is how we will pass the model to the activities that will notify the host about events in the workflow.
  3. The workflow accepts two arguments

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| CountTo | Int32 | The number you want to count to |
| CountDelay | Int32 | The number of milliseconds you want to delay after each count |

* 1. 
  2. Figure 4
  3. The Workflow that will do the counting and notifications
  4. As you can see in the workflow I've created a number of custom activities that do notifications when the count starts, completes, cancels and updates. To handle the case where the workflow is canceled I use a CancellationScope activity which allows me to specify a CancellationHandler for the notification.
  5. The notification activities are surprisingly simple
  6. C#
  7. public sealed class NotifyCountUpdated : NativeActivity
  8. {
  9. public InArgument<Int32> CurrentCount { get; set; }
  11. protected override void CacheMetadata(NativeActivityMetadata metadata)
  12. {
  13. metadata.RequireExtension<CountModel>();
  14. var currentCountArg = new RuntimeArgument(
  15. "CurrentCount", typeof(Int32), ArgumentDirection.In);
  16. metadata.AddArgument(currentCountArg);
  17. metadata.Bind(this.CurrentCount, currentCountArg);
  18. }
  19. protected override void Execute(NativeActivityContext context)
  20. {
  21. var countModel = context.GetExtension<ICountModel>();
  23. if (countModel.CountUpdated != null)
  24. {
  25. countModel.CountUpdated(
  26. this.CurrentCount.Get(context));
  27. }
  28. }
  29. }
  30. The activity obtains the model from the extensions collection and if the notification delegate is provided the activity invokes it.

### Starting the Workflow

* 1. When I want to invoke the workflow I have to add the extension before running the workflow. In this case, the model is the extension so I simply add "this".
  2. C#
  3. public void StartCounting(int countTo = 100, int countDelay = 50)
  4. {
  5. this.countTo = countTo;
  6. this.countDelay = countDelay;
  7. workflow = new WorkflowApplication(
  8. new WorkflowCount()
  9. {
  10. CountTo = this.countTo,
  11. CountDelay = this.countDelay
  12. });
  13. workflow.Extensions.Add(this);
  14. workflow.Run();
  15. }

## Canceling the Workflow

To cancel the workflow there are two cases. In one case we want to cancel the workflow synchronously as in the case where the user clicked the button to cancel.

* 1. C#
  2. public void CancelCounting()
  3. {
  4. workflow.Cancel();
  5. }

The other case is when the WPF window is closing. In this case we need to cancel the workflow but we want to do it asynchronously. If you cancel synchronously in the Window.Closing event the window will appear to freeze momentarily when closing. Instead what you want is to start the cancel process in Window.Closing and then finish it in the Window.Closed event. Usually the cancel will be complete by the time the Closed event is fired. Even if it isn't the window won't be visible while it tries to cancel.

Even though the model doesn't need this capability, the ViewModel will need to handle the closing/close events and it doesn't have access to the workflow so the model will need to expose these methods.

* 1. C#
  2. public IAsyncResult BeginCancelCounting(
  3. AsyncCallback callback, object state)
  4. {
  5. return this.workflow != null
  6. ? this.workflow.BeginCancel(callback, state)
  7. : null;
  8. }
  9. public void EndCancelCounting(IAsyncResult result)
  10. {
  11. if (workflow != null && result != null)
  12. workflow.EndCancel(result);
  13. }

How to Test the Model

* 1. At this point there are two things that need testing. I built the tests as I went along and here is how I did it.

### Testing the Notification Activities

* 1. There are the following aspects to the notification activities that I want to test.
  2. Do they respond appropriately if the extension is not present (throwing an exception)?
  3. Do they raise the notification if the extension is present and the delegate is supplied?
  4. Do they fail if the extension is present but no delegate is supplied?
  5. For each of the Notification Activities I have tests that verify the behavior.

#### Verifying that the activity throws an exception if the extension is not found

* 1. C#
  2. [TestMethod]
  3. [ExpectedException(typeof(ValidationException))]
  4. public void NotifyStartedShouldThrowOnNoExtension()
  5. {
  6. WorkflowInvoker.Invoke(new NotifyCountStarted());
  7. }

### Verifiying that the activity raises the notification if the extension is present and the delegate is supplied

* 1. C#
  2. [TestMethod]
  3. public void NotifyUpdatedShouldNotify()
  4. {
  5. ICountModel model = CountModelFactory.CreateModel();
  6. bool updateNotified = false;
  7. int actual = 0;
  8. int expected = 32;
  9. // The delegate will be invoked with the count
  10. model.CountUpdated = (i) =>
  11. {
  12. updateNotified = true;
  13. actual = i;
  14. };
  15. WorkflowInvoker invoker = new WorkflowInvoker(
  16. new NotifyCountUpdated()
  17. {
  18. CurrentCount = expected
  19. });
  20. invoker.Extensions.Add(model);
  21. invoker.Invoke();
  22. Assert.IsTrue(updateNotified);
  23. Assert.AreEqual(expected, actual);
  24. }

### Verifying that the activity does not fail if the extension is present but no delegate is supplied

* 1. C#
  2. [TestMethod]
  3. public void NotifyUpdatedWithNoDelegateShouldDoNothing()
  4. {
  5. ICountModel model = CountModelFactory.CreateModel();
  6. WorkflowInvoker invoker = new WorkflowInvoker(new NotifyCountUpdated());
  7. invoker.Extensions.Add(model);
  8. invoker.Invoke();
  9. // No exception is success

}

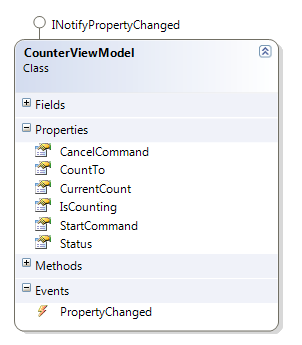
### Testing the Model

When testing the model I have to take into account the WorkflowApplication will run the workflow on a different thread. That means I will need to use synchronization objects to cause the test thread to wait.

Creating the tests while I was developing the class forced me to deal with this reality the Action delegates really helped here.

* 1. C#
  2. [TestMethod()]
  3. public void ShouldRaiseCompleteEvent()
  4. {
  5. ICountModel target = CountModelFactory.CreateModel();
  6. bool completeEventRaised = false;
  7. AutoResetEvent countCompleted = new AutoResetEvent(false);
  8. target.CountCompleted = () =>
  9. {
  10. completeEventRaised = true;
  11. countCompleted.Set();
  12. };
  13. target.StartCounting(1);
  14. countCompleted.WaitOne(1000);
  15. Assert.IsTrue(completeEventRaised);
  16. }

Implementing the ViewModel

* 1. The purpose of the view model is to provide the specifics that a particular View needs. My goal in creating the view was to create a "bind-able" surface that would allow the controls in the WPF application to data bind to both data and commands.
  2. 
  3. Figure 5
  4. The Bind-Able surface of the CounterViewModel class
  5. Being new to WPF I found the most challenging aspect of getting this done was the commands. I found a sample application on MSDN that includes a class called **RelayCommand** that allows my view to specify delegates that the command will invoke.
  6. Binding to the commands causes the buttons on the form to be enabled and disabled as the commands are enabled and disabled. When you bind to data, updates are provided through the PropertyChanged event. However, when you update the state of a command you need to use CommandManager.InvalidateRequerySuggested. I found that when I invoked this method on a callback from the workflow the command state was not updated. To solve this problem I created a method to dispatch the call to the UI thread.
  7. C#
  8. private static void RequeryCommands()
  9. {
  10. // May be called at shutdown
  11. if (Application.Current != null)
  12. {
  13. Application.Current.Dispatcher.Invoke(
  14. (Action)(() => CommandManager.InvalidateRequerySuggested()));
  15. }
  16. }
  17. The rest of the code in the ViewModel is pretty straightforward.

### Binding to the ViewModel

* 1. DataBinding in XAML can be tricky at first but here is how I did it. In the MainWindow constructor I create the view model, set the DataContext to the view model and setup event handlers to solve the window closing problem.
  2. C#
  3. public MainWindow()
  4. {
  5. this.InitializeComponent();
  7. this.viewModel = new CounterViewModel();
  8. this.Closing += this.viewModel.ViewClosing;
  9. this.Closed += this.viewModel.ViewClosed;
  11. this.DataContext = this.viewModel;
  12. }
  13. This is all the code in the code behind for the MainWindow class. It is difficult to test code in the window class so this is a good thing.
  14. Here is how I did the databinding for the progress bar.
  15. XAML
  16. <ProgressBar
  17. x:Name="progressBar"
  18. Grid.Column="0"
  19. Grid.Row="2"
  20. Grid.ColumnSpan="2"
  21. Value="{Binding Path=CurrentCount, Mode=OneWay}"
  22. Maximum="{Binding Path=CountTo, Mode=OneWay}"
  23. />
  24. Binding to commands is similar
  25. XAML
  26. <Button
  27. FontSize="32"
  28. Grid.Row="1"
  29. Grid.Column="0"
  30. Command="{Binding Path=StartCommand}"
  31. >Start</Button>

### Testing the ViewModel

* 1. Initially the ViewModel didn't include delegates that surface the notification events from the model because I didn't write the tests first and instead created the UI. In retrospect I should have created the tests at the same time as I created the View Model. I would have realized that it was impossible to test the View Model without the events because the test code has to know about the events in the workflow.
  2. For example, here is a test where I want to verify the state of the StartCommand
  3. C#
  4. /// <summary>
  5. /// Verifies that the StartCounting command is enabled/disabled as expected
  6. ///</summary>
  7. [TestMethod()]
  8. public void CanStartCountingTest()
  9. {
  10. CounterViewModel target = new CounterViewModel();
  11. AutoResetEvent countStarted = new AutoResetEvent(false);
  12. AutoResetEvent countCompleted = new AutoResetEvent(false);
  13. target.CountStarted += () => countStarted.Set();
  14. target.CountCompleted += () => countCompleted.Set();
  15. Assert.IsTrue(target.CanStartCounting(), "Start command is not enabled");
  16. target.CountTo = 5;
  17. target.StartCounting();
  18. Assert.IsTrue(countStarted.WaitOne(1000));
  19. Assert.IsFalse(target.CanStartCounting(), "Start command is not disabled");
  20. Assert.IsTrue(countCompleted.WaitOne(1000));
  21. Assert.IsTrue(target.CanStartCounting(), "Start command is not enabled");
  22. }

Summary

* 1. This sample application demonstrates that it is possible to build a WPF app using the MVVM pattern that interacts with a workflow. Using a Workflow to do processing makes your app more responsive because the workflow is running on a different thread. Also, though this app does not show it, you could run many workflows all at the same time by interacting with a collection of them rather than a single one like this application does.