**Architecture overview of the Workflow Manager**

First let me start with a bit of history. Before joining the WF team I was a dev on the Biztalk tracking/management team and so I had some ideas on how to design user experience around managing workflows. I have started working on the app back in December 2004 and originally it had two processes – one hosting the Runtime and the other – UI application. I was using .Net Remoting as communication link between the two. Later I got rid of the Remoting to simplify life a bit (both the runtime and the UI now run in the same process) but some of that stuff is still in the code, namely the AdministrationService and the AdministrationServiceProxy classes that are being used on the host/UI sides respectively to channel events and calls between the two.

Now let get into the architecture overview and then will drill into interesting pieces one by on in the later posts. As I said before the sample consist of two forms – one hosts the WF runtime (WinOEHost) and the other is the main client app form (WorkflowManager).

The list view of the main client form (DataGridViewExplore) is a subclass of the new managed DataGridView control. It custom paints all list entries in the Office Outlook two-line format. The data is being provided by the WorkflowInstanceManager which implements IBindingList interface to dynamically notify its clients about changes in the list. Based on the user selection in the Workflows tree view (either Live or Tracked), the WorkflowInstanceManager would pull the data from either the AdministrationServiceProxy or SQLTrackingDataProvider, both of which implement IDataProvider. In case of the Live data selected, it would also sink state change events (which include instance create/completed/terminated/etc and activity started/completed/etc). Both data providers return a list of IWorkflowInstance objects.

Let’s drill into the Live data case a little deeper. AdministrationService uses a custom tracking service TrackingEventsHelper that in its tracking profile asks for all activity state change events for all activities in the workflow. These events are then cached in memory for every live workflow in the corresponding instance of the LiveWorkflowInstance class. This class is a wrapper around the WF’s WorkflowInstance class and adds ability to 1) store and return by a request a list of prior activity state change events, 2) serialize current workflow definition into markup (with some workarounds for the custom root activity case) and 3) simplifies applying a batch of workflow changes supplied by the UI client.

The details view uses the ViewHost control to show workflow in a graphical form. There is a little trick in the WorkflowDesignSurface to have the design surface in the read-only or read-write form – take a look at MakePropertiesReadOnly (IServiceProvider, object) function to see how it’s done.

The UI-driven dynamic updates are enabled by the DynamicUpdateDriver class that sinks ComponentChanged events and adds an instance of the ActivityDynamicChangeLog for every component changed action performed by the user.

**Instance state and per-instance Title property handling**

Let’s look at how the app keeps track of what’s going on inside the runtime and the individual workflow instances.

All this functionality is factored into the WorkflowAdminService project, AdministrationService class.

This class has two sources of events – runtime itself (the WorkflowStarted, WorkflowSuspended, WorkflowCompleted, and WorkflowTerminated events) and a custom tracking service (TrackingEventsHelper class).

The tracking service creates a tracking profile for each workflow type

private TrackingProfile GetDefaultProfile(Type scheduleType)

in which it essentially asks for all activity execution status events on all activities and all workflow events:

      TrackingProfile profile = new TrackingProfile();

      profile.Version = new Version("1.0.0");

      //all activities, all state changes

      ActivityTrackPoint atp = new ActivityTrackPoint();

      ActivityTrackingLocation location = new ActivityTrackingLocation(typeof(Activity));

      location.MatchDerivedTypes = true;

      foreach (ActivityExecutionStatus s in Enum.GetValues(typeof(ActivityExecutionStatus)))

            location.ExecutionStatusEvents.Add(s);

      atp.MatchingLocations.Add(location);

      profile.ActivityTrackPoints.Add(atp);

      // Add a TrackPoint to receive all workflow status events

      WorkflowTrackPoint workflowTrackPoint = new WorkflowTrackPoint();

      workflowTrackPoint.MatchingLocation = new WorkflowTrackingLocation();

      foreach (TrackingWorkflowEvent workflowEvent in Enum.GetValues(typeof(TrackingWorkflowEvent)))        workflowTrackPoint.MatchingLocation.Events.Add(workflowEvent);

      profile.WorkflowTrackPoints.Add(workflowTrackPoint);

In addition, if the workflow type has a property called Title, the tracking profile would have a request to extract a value of that property when the workflow enters the executing state (to make sure

protected override void Initialize (IServiceProvider)

has been called already):

      ActivityTrackPoint atp = new ActivityTrackPoint();

      ActivityTrackingLocation location = new ActivityTrackingLocation(scheduleType, new ActivityExecutionStatus[] { ActivityExecutionStatus.Executing});

      atp.MatchingLocations.Add(location);

      ActivityTrackingCondition matchingActivityTrackingCondition = new ActivityTrackingCondition("Name", rootActivity.Name);

      matchingActivityTrackingCondition.Operator = ComparisonOperator.Equals;

      location.Conditions.Add(matchingActivityTrackingCondition);

      WorkflowDataTrackingExtract workflowDataTrackingExtract = new WorkflowDataTrackingExtract();

      workflowDataTrackingExtract.Member = "Title";

      atp.Extracts.Add(workflowDataTrackingExtract);

      profile.ActivityTrackPoints.Add(atp);

The Initialize() method on the workflow class would set the title property to some instance specific value (e.g. in a document approval scenario it could be the document name along with the deadline and approver names).

Here is a simple example on how the title property could be declared and set:

public sealed partial class Workflow1 : SequentialWorkflowActivity

{

      private string title = string.Empty;

      public string Title

      {

            get { return this.title; }

      }

      protected override void Initialize(IServiceProvider provider)

      {

            base.Initialize(provider);

            this.title = "Some instance specific value.";

      }

}

When the tracking service receives a tracking event, it analyzes the tracking record and if it’s an activity tracking record, extracts the value of the title property:

      ActivityTrackingRecord activityTrackingRecord = record as ActivityTrackingRecord;

      if (activityTrackingRecord != null)

      {

            if (activityTrackingRecord.Body != null && activityTrackingRecord.Body.Count > 0)

            {

                  TrackingDataItem trackedItem = activityTrackingRecord.Body[0];

                  if (trackedItem.FieldName.Equals("Title", StringComparison.Ordinal))

                  {

                        string title = trackedItem.Data as string;

                  }

            }

      }

This title is then shown in the UI (in the instance list and on the details view) so that the user could distinguish between different instances of the same workflow type.

**Workflow Changes (aka Dynamic Updates) Part one: UI-based changes**

Today I’ll go into more details on how to apply the changes based on the user input from the designer surface.

As I said in one of the previous posts, it is the DynamicUpdateDriver class that sinks the ComponentChanged event from the IComponentChangeService, and in the corresponding event handler it analyzes the passed arguments in the following way:

void componentChangeService\_ComponentChanged(object sender, ComponentChangedEventArgs e)

{

      Activity modifiedActivity = e.Component as Activity;

      if (modifiedActivity != null)

      {

            if (e.NewValue == null && e.OldValue != null)

            {

                  ActivityCollectionChangeEventArgs collectionChangedArgs = e.OldValue as ActivityCollectionChangeEventArgs;

                  CompositeActivity owner = collectionChangedArgs.Owner as CompositeActivity;

                  if (collectionChangedArgs.RemovedItems.Count != 0 && collectionChangedArgs.AddedItems.Count == 0)

                  {

                        //deletion

                  }

                  else if (collectionChangedArgs.RemovedItems.Count == 0 && collectionChangedArgs.AddedItems.Count != 0)

                  {

                        //addition

                  }

            }

            else if (e.NewValue != null && e.OldValue != null)

            {

                  //modification

            }

      }

}

Based on the kind of changes it creates an instance of the ActivityDynamicChangeLog class and stores it along with other logged changes into a list. When the user submits the changes to the runtime, the ApplyDynamicUpdateBatch () iterates over the changes and applies them to the Transient Workflow in the same order they were performed (note that this.instance is a member variable of type WorkflowInstance):

Activity currentRoot = this.instance.GetWorkflowDefinition();

workflowChanges = new WorkflowChanges(currentRoot);

CompositeActivity root = workflowChanges.TransientWorkflow;

foreach (ActivityDynamicChangeLog change in dynamicUpdateCommitData.ChangeLog)

{

    if (change.ChangeType == ActivityDynamicUpdateType.Add)

    {

      …

    }

    else if (change.ChangeType == ActivityDynamicUpdateType.Modify)

    {

      …

    }

    else if (change.ChangeType == ActivityDynamicUpdateType.Remove)

    {

      …

    }

}

this.instance.ApplyWorkflowChanges(workflowChanges);

And that's it!

**Workflow Changes Part two: workarounds for the custom activities with locked children**

If your custom activity/workflow has children then they will be locked and you would not be able to add/remove those when re-using the activity in other activities/workflows. That means that a custom activity writer can safely assume his/her activity would be used as it is without changers. On the other hand it also means that once you create your custom activity, you can not really change it through regular means at either design or run time.

Besides these “by design” constrains there are some issues around the locked activities that have not been completly resolved in WF v1.0:

1)      WF serializer doesn’t serialize dynamically updated instances with locked activities correctly (it assumes that the custom activity would initialize itself correctly and thus outputs just the top level activity into xoml).

2)      Once the instance has been initialized, you can not change it at design time. This means that if you open an instance of such activity in workflow view, you would not be able to add/delete activities in it as well as change any of the properties on the activity or any of it’s children. What it also means, when you open an activity tree from currently executing instance, you’d not be able to construct the changes visually without some workaround.

Lets first look into the second issue. If you create a new workflow control library project in VS and open up the workflow1.designer.cs file, you’d see that code inside InitializeComponent() is wrapped by two CanModifyActivities property sets:

private void InitializeComponent()

{

      this.CanModifyActivities = true;

      ...

      // add activities, set their properties, etc...

      ...

      this.CanModifyActivities = false;

}

The CanModifyActivities is a protected property defined on the CompositeActivity class. The workaround that I use in the Workflow manager is to set this property to true when I want to be able to modify the custom activity with locked children in the workflow view for applying the changes to the runtime instance as discussed in the part one of this post:

internal void UpdateCanModifyFlag(IList<Activity> activities)

{

      if (!AllowDynamicUpdates)

            return;

      List<Activity> processingActivities = new List<Activity>();

      foreach (Activity rootActivity in activities)

      {

            processingActivities.Add(rootActivity);

            CompositeActivity compositeActivity = rootActivity as CompositeActivity;

            if(compositeActivity != null)

                  processingActivities.AddRange(GetNestedActivities(compositeActivity));

      }

      foreach (Activity childActivity in processingActivities)

      {

            if (childActivity is CompositeActivity)

            {

                  //this will set the dynamic updates flag on the root activity

                  Type activityType = childActivity.GetType();

                  PropertyInfo readOnlyProperty = activityType.GetProperty("CanModifyActivities", BindingFlags.NonPublic | BindingFlags.Instance);

                  readOnlyProperty.SetValue(childActivity, true, null);

            }

      }

}

Not the nicest and fastest piece of code, but it works.

The first issue will be solved in the upcoming third part of the article, stay tuned!

**Workflow Changes Part Three: tracking stream based**

If you have instance data coming from the tracking database and want to get the actual instance activity tree after the changes were applied, it gets a little tricky.

First, you’d need to

a)      read all of the activity add/remove changes from the database and merge them into single list

b)      order them correctly

To complicate the matters a little bit, the changes are split into two tables, AddedActivity and RemovedActivity. The app uses two stored procs, [dbo].[GetAddActivityActionsForWorkflowInstance]

and

[dbo].[GetRemovedActivityActionsForWorkflowInstance] to read those.

Every time workflow changes are applied, they are tracked as a single instance event.

But since there may be multiple activities added/removed in a single tracked change, they need to be applied in the order they were made during every change. Let’s say that during single workflow change session you first added a parallel activity, then removed one of the branches and then dropped a code activity into the other branch. Now, if you apply those the changes in a wrong order, you’d simply fail.

That’s why after added/removed changes are read, they need to be merged into a single list and then sorted by the two properties, WorkflowInstanceEventId and Order.

Here is a little neat class that does the sorting (which got enabled by the new Whidbey List<> class):

private sealed class TrackedChangeActionSorter : IComparer<TrackedChangeAction>

{

      public static void Sort(List<TrackedChangeAction> actions)

      {

            TrackedChangeActionSorter sorter = new TrackedChangeActionSorter();

            actions.Sort(sorter);

      }

      int **IComparer<TrackedChangeAction>.Compare**(TrackedChangeAction lhs, TrackedChangeAction rhs)

      {

            if (lhs.WorkflowInstanceEventId < rhs.WorkflowInstanceEventId)

                  return -1;

            else if (lhs.WorkflowInstanceEventId == rhs.WorkflowInstanceEventId)

            {

                  if (lhs.Order < rhs.Order)

                        return -1;

                  else if (lhs.Order == rhs.Order)

                        return 0;

                  else

                        return +1;

            }

            else

                  return +1;

      }

}

Now that you have the changes in the right order, they need to be applied to the original activity tree. As you may have noticed, some of the changes have non-null AddedActivityAction or RemovedActivityAction properties. If not null, they hold markup-serialized instances of ActivityChangeAction–derived class (either AddedActivityAction or RemovedActivityAction).

Once you got hold of those, you may deserialize and apply them (in the correctly sorted order) to the actual activity tree. The method we need is unfortunately marked protected in WF v1 so we’d need to use reflection to call it:

CompositeActivity rootActivity = ...;

WorkflowChangeAction[] changes = ...;

if (rootActivity != null && changes != null && changes.Length > 0)

{

      for (int i = 0; i < changes.Length; i++)

      {

            WorkflowChangeAction changeAction = changes[i];

            if(changeAction != null)

            {

                  //protected abstract bool ApplyTo(Activity rootActivity);

                  MethodInfo applyMethod = changeAction.GetType().GetMethod("ApplyTo", BindingFlags.Instance | BindingFlags.NonPublic);

                  applyMethod.Invoke(changeAction, new object[] { rootActivity });

            }

      }

}

Another neat class I wanted to point out is the ActivityMarkupWithChanges. In v1 we have a limitation in the serializer that doesn’t allow serializing changed custom activity/workflow types with children activities in them (see issue number one in the second part of this article). The mentioned class was created to overcome that limitation. What it does it serializes list of the applied changes (stored in a form of WorkflowChangeAction-derived instances) along with the original workflow:

[Serializable]

public class ActivityMarkupWithChanges

{

      private Activity root = null;

      private WorkflowChangeAction[] changes = null;

      public ActivityMarkupWithChanges()

      {

      }

      public ActivityMarkupWithChanges(Activity root, WorkflowChangeAction[] changes)

      {

            this.root = root;

            this.changes = changes;

      }

      public Activity Root

      {

            get { return this.root; }

            set { this.root = value; }

      }

      public WorkflowChangeAction[] Changes

      {

            get { return this.changes; }

            set { this.changes = value; }

      }

}

As a matter of fact the code snipped above with an example of ApplyTo usage is exactly what the:

protected override void PerformLoad(IDesignerSerializationManager iSerializationManager)

does after it deserializes an instance of ActivityMarkupWithChanges with non-empty list of changes.