.NET Programming

Windows Workflow Foundation

WF

B. Bukovics, Pro WF. Windows Workflow in .NET 3.5, Apress 2008
T. Kitta, Professional Windows Workflow Foundation, Wrox, 2007
B. Myers, Foundation of WF. An Introduction to Windows Workflow Foundation, Apress, 2006
MSDN

Krzysztof Mossakowski
Windows Workflow Foundation Overview

- Windows Workflow Foundation is the programming model, engine, and tools for quickly building workflow-enabled applications on Windows.
- It is a freely distributable .NET component (as a part of .NET Framework 3.0 and later).
  - It also ships as a part of Windows Vista and newer.
- It contains:
  - A namespace
  - An in-process workflow engine
  - Designers for Visual Studio.
Workflows

- A workflow is a set of elemental units called **activities** that are stored as a model that describes a real-world process.
- Workflows provide a way of describing the order of execution and dependent relationships between pieces of short- or long-running work.
- This work passes through the model from start to finish, and activities might be executed by people or by system functions.
Workflow Types

- **Sequential workflows** (derived from the `SequentialWorkflowActivity` class)
  - They declare a series of steps that are executed in a prescribed order
  - The workflow has a defined beginning and end

- **State machine workflows** (derived from the `StateMachineWorkflowActivity` class)
  - They define a set of states, with possible transitions between each state
  - A state machine workflow is not constrained by a static sequence of steps
  - Execution does not always have to begin with tasks in the first state
Sequential Workflow Example
State Machine Workflow Example

Workflow Runtime Engine

- Every running workflow instance is created and maintained by an in-process runtime engine.
- There can be several workflow runtime engines within an application domain, and each instance of the runtime engine can support multiple workflow instances running concurrently.
- When a workflow model is compiled, it can be executed inside any Windows process including: console applications, forms-based applications, Windows Services, ASP.NET Web sites, and WCF services.
- A core runtime engine is represented by the `System.Workflow.Runtime.WorkflowRuntime` class.
  - An instance of this class must be hosted in the application in order to execute and manage workflows.
WorkflowRuntime Events

- WorkflowCreated
- WorkflowStarted
- WorkflowIdled – when idled by the runtime engine
- WorkflowAborted
- WorkflowPersisted
- WorkflowUnloaded, WorkflowLoaded
- WorkflowSuspended, WorkflowResumed
- WorkflowTerminated – when abnormally terminated (should always be subscribed)
- WorkflowCompleted – when completed normally (should always be subscribed)
Application Host Process

- WF is not provided as a complete stand-alone application
- A host application that is responsible for creating an instance of the workflow runtime engine (the `WorkflowRuntime` class) must be provided, e.g.:
  - Console application
  - Windows service
  - WinForms application
  - WPF application
  - ASP.NET web application
- Workflow instances can be also exposed as WCF services
  - The `System.ServiceModel.WorkflowServiceHost` class combines the basic capabilities of `WorkflowRuntime` with `ServiceHost`
Simple Application Host Example

```csharp
static void Main(string[] args)
{
    using (WorkflowRuntime workflowRuntime = new WorkflowRuntime()) {
        AutoResetEvent waitHandle = new AutoResetEvent(false);
        workflowRuntime.WorkflowCompleted +=
            delegate (object sender, WorkflowCompletedEventArgs e)
            {
                waitHandle.Set();
            };
        workflowRuntime.WorkflowTerminated +=
            delegate (object sender, WorkflowTerminatedEventArgs e)
            {
                Console.WriteLine(e.Exception.Message);
                waitHandle.Set();
            };

        WorkflowInstance instance = workflowRuntime.CreateWorkflow(
            typeof(WorkflowConsoleApplication1.Workflow1));
        instance.Start();

        waitHandle.WaitOne();
    }
}
```
Services

- WF provides default implementations of the runtime services that meet the needs of many types of applications, e.g.:
  - Persistence
  - Scheduling
  - Transaction
  - Tracking
- Service components are pluggable
- Custom services can be created to extend the Windows Workflow Foundation platform by deriving from the base service classes
Persistance

- Durability of workflow is a key feature of WF
  - The workflow runtime engine manages workflow execution and enables workflows to remain active for long periods of time and survive application restarts

- Workflows can be unloaded from memory while awaiting input and serialized into a persistent store
  - SQL database or XML file can be used as a store
  - Whenever the input is received, the workflow runtime engine loads the workflow state information back into memory and continues execution of the workflow

- Available services:
  - SqlWorkflowPersistanceService for SQL Server
  - Custom, derived from the WorkflowPersistanceService class
Tracking

- Tracking is the ability to specify and capture information about workflow instances.

- When a new workflow is created, the tracking service requests a tracking channel which will be used as a destination for the tracking information.

- The tracking service can track three types of events: Workflow instance, Activity, and User.
  - The type of amount of information can be configured.

- Available services:
  - SqlTrackingService for SQL Server
  - Custom, derived from the TrackingService class
Local Communication and Correlation

- Host processes can communicate with workflows by exchanging data through custom local communication services
  - These local communication services implement user-defined interfaces that defines methods and events that will be passed between the workflow and the host process
- Correlation is an interaction of host processes with a specific activity in a specific workflow instance using a unique ID
  - ID passed between the host process and the workflow as an event argument
Workflow Changes

- WF allows to dynamically update a workflow instance and declarative rules during run time
  - E.g. the expected behaviours, flow control, etc. can be changed before activities are scheduled for execution
- Business processing logic can be modified without having to recompile and restart a workflow
Workflow Authoring Modes

- WF supports three different ways to author workflows:
  - Code-only
    - when a workflow is designed, the WF designer generates code and places it in the Workflow1.designer.cs file
  - Code-separation
    - a serialized form of the workflow model is saved in a markup file named Workflow1.xoml
    - the code is saved in a separate file named Workflow1.xoml.cs
  - No-code
    - workflows are defined entirely in a .xoml file
    - the integrated Visual Studio tools for WF don’t directly support the no-code authoring of workflows
Workflow Markup

- Workflow markup is based on XAML (extensible Application Markup Language)
  - For historical reasons, the file extensions are still .xoml, but internally, they use the XAML syntax
- It allows developers and designers to model business logic declaratively
  - Lower-level implementation details are modelled by code-beside files
- It is possible to activate a workflow by directly loading a workflow markup file into the workflow runtime engine at run time
Workflow Markup Example

```xml
<SequentialWorkflowActivity x:Class="XAMLWorkflow.Workflow1"
    x:Name="Workflow1" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <IfElseActivity x:Name="ifElseActivity1">
    <IfElseBranchActivity x:Name="ifElseBranchActivity1">
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="EvalCondition" />
      </IfElseBranchActivity.Condition>
      <CodeActivity x:Name="codeActivity1"
        ExecuteCode="codeActivity1_ExecuteCode" />
    </IfElseBranchActivity>
    <IfElseBranchActivity x:Name="ifElseBranchActivity2">
      <CodeActivity x:Name="codeActivity2"
        ExecuteCode="codeActivity2_ExecuteCode" />
    </IfElseBranchActivity>
  </IfElseActivity>
</SequentialWorkflowActivity>
```
Workflow Instance

- A workflow instance is a single executable instance of a workflow.
- It is represented in code by the `WorkflowInstance` class:
  - This class has public members that permit to interact and control that instance of a workflow.
- To create a workflow instance, call the `WorkflowRuntime.CreateWorkflow` method:
  - It returns a `WorkflowInstance` object.
- To start execution of the workflow instance, call the `WorkflowInstance.Start` method.
- The `WorkflowInstance.InstanceId` property is the unique ID of the workflow instance.
Activities

- Activities are the elemental unit of a workflow
- When all the activities in a given flow path are finished running, the workflow instance is completed
- Activities have two types of behaviour:
  - The runtime behaviour specifies the actions upon execution
  - The design time behaviour controls the appearance of the activity and its interaction while being displayed within the designer
- Windows Workflow Foundation contains a library of standard activities and provides the mechanisms to create custom activities
Activities Class Hierarchy

- Activity
  - /standard simple activities/
  - **CompositeActivity** (can host other activities)
    - /standard composite activities/
  - **StateActivity** (represents a state in a state machine workflow)
    - **StateMachineWorkflowActivity** (a base for all state machine workflows)
  - **SequenceActivity** (supports the execution of an ordered set of child activities)
    - **SequentialWorkflowActivity** (a base for all sequential workflows)
Standard Activities

- The major activity categories of standard activities are the following:
  - Custom workflow logic
  - Flow control
  - State management
  - Event handling
  - Local service communication
  - Rules
  - Web services
  - Workflow services
  - Transactions, compensation, and synchronization
  - Exceptions and error handling
Custom Workflow Logic Activities

- **CodeActivity**
  - It is a simple way to add code to a workflow.
  - The handler code for the **ExecuteCode** event can perform any custom work.
  - It can be thought of as a lightweight custom activity, but it is reusable only within that workflow.
Flow Control Activities

- **IfElseActivity** and **IfElseBranchActivity**
  - The **IfElseActivity** permits to declare multiple conditional branches of execution within a workflow.
  - The **IfElseActivity** is a composite activity that contains two or more instances of **IfElseBranchActivity**

- **WhileActivity**
  - It repeatedly executes a single child activity as long as its condition is true

- **DelayActivity**
  - It is used to temporarily halt execution of a workflow for a specified amount of time or until receiving an external event
Flow Control Activities cont.

- **SequenceActivity**
  - It allows to add multiple child activities that are executed in a defined order.
  - It is especially useful as a child of other activities that accept only a single child (e.g. `WhileActivity`, `ReplicatorActivity`).

- **ParallelActivity**
  - It allows to schedule two or more child branches for execution at the same time.
  - Each workflow instance executes on a single thread, so true parallel execution of multiple activities isn’t possible.
Flow Control Activities cont.

- **ReplicatorActivity**
  - It creates and executes multiple copies of a single child activity
  - If the single child activity is a composite, all of the activities within the composite are executed for each object in the collection
  - It supports sequential and parallel execution of the child activity (for sequential execution, only one instance of the child activity is created and executed at any one time)
  - It doesn’t actually execute multiple child activities simultaneously
Flow Control Activities cont.

- **ConditionedActivityGroup**
  - It is used to conditionally execute a series of child activities; each child activity is associated with its own *WhenCondition* (the child activity continuously executes as long this property is true)
  - It also has an optional *UntilCondition* property that defines a condition used to stop execution

- **InvokeWorkflowActivity**
  - It is used to start a second workflow
  - The second workflow executes asynchronously, so this activity will return before the new workflow completes
Flow Control Activities cont.

- **SuspendActivity**
  - It stops the execution of the workflow but doesn’t permanently terminate the workflow.
  - To restart a suspended workflow, call the `Resume` method on the `WorkflowInstance` object.

- **TerminateActivity**
  - It immediately stops the execution of a workflow instance.
  - A descriptive error message can be provided with the termination by setting the `Error` property of the `TerminateActivity`
State Management Activities

- **StateActivity**
  - It is the primary building block of all state machine workflows
  - It represents a single state

- **SetStateActivity**
  - It is used to make transition to a different state
  - The SetStateActivity is usually declared as a child of an EventDrivenActivity. The **EventDrivenActivity** is used within a StateActivity to wait for an external event

- **StateInitializationActivity** and **StateFinalizationActivity**
  - They are optional activities that contain activities to execute when a state is first entered and just before transition to another state resp.
Event Handling Activities

- **ListenActivity** and **EventDrivenActivity**
  
  - The ListenActivity is a composite activity that contains two or more instances of EventDrivenActivity.
  
  - Each EventDrivenActivity defines an execution branch that is waiting for a different event.
  
  - The EventDrivenActivity contains its own set of child activities that are executed when the event is received.
  
  - Only one EventDrivenActivity branch is ever executed; when one branch receives its event, the other branches are canceled.
  
  - The EventDrivenActivity is often used as a child of the StateActivity in a state machine workflows.
Event Handling Activities cont.

- **EventHandlersActivity** and **EventHandlingScopeActivity**
  - They are used in advanced scenarios where there is a need to concurrently handle multiple events.
  - The EventHandlersActivity is a container for multiple instances of the **EventDrivenActivity**.
  - The EventHandlersActivity does not cancel the unused branches (as opposed to the **ListenActivity**).
  - The EventHandlingScopeActivity can act as the parent of the EventHandlersActivity and permits to also define a set of mainline activities to execute.
Local Service Communication Activities

- **CallExternalMethodActivity**
  - It is used to synchronously invoke a method implemented in a local workflow service.
  - The method that is invoked must be defined in an interface, and that interface must be decorated with the `ExternalDataExchangeAttribute`.
  - Parameters can be passed using the `ParameterBindings` collection.

- **HandleExternalEventActivity**
  - It is used to receive an event from a local service.
  - The `HandleExternalEventActivity` is never used by itself. It must be contained within an `EventDrivenActivity`, and it must be the first child activity.
Rules Activity

- **PolicyActivity**
  - It allows to define a set of rules that act upon fields or properties in a workflow
  - It is designed to directly change workflow state by allowing to update field and property values or execute methods
Web Services Activities

- **InvokeWebServiceActivity**
  - It is used to invoke a web service from within a workflow

- **WebServiceInputActivity**
  - It exposes the workflow as a web service method

- **WebServiceOutputActivity**
  - It is used along with the WebServiceInputActivity to expose a workflow to web service clients
  - It is responsible for sending a response to the web service client

- **WebServiceFaultActivity**
  - It is used when you need to send a web service fault to a client to indicate an error condition
Workflow Services Activities

- **ReceiveActivity**
  - It is used to expose a workflow as a WCF service
  - WF supports two authoring modes for WCF services:
    - **Contract-first** authoring is used when the WCF service contract is designed first
    - **Workflow-first** authoring is used when a service contract is derived from the implementation of the workflow

- **SendActivity**
  - It is used to invoke an operation on a WCF service declaratively from within a workflow
  - The WCF service contract must exist prior to using the SendActivity
Transactions and Compensation Activities

- **TransactionScopeActivity**
  - It is used to define a logical unit of work that succeeds or fails in its entirety
  - When all child activities within the TransactionScopeActivity complete normally, the transaction is closed, and all pending work is committed

- **CompensatableTransactionScopeActivity**
  - It is similar to the TransactionScopeActivity, but it also supports compensation

- **CompensatableSequenceActivity**
  - It is a version of the *SequenceActivity* that supports compensation
Transactions and Compensation Activities

- **CompensateActivity**
  - It allows to manually control the compensation process and directly execute the compensation logic for an activity
  - (Normally, compensation is triggered when an unhandled exception is thrown within a workflow)

- **CompensationHandlerActivity**
  - It is the container for activities that should execute with compensation support
  - This activity can be added only to an activity that supports the `ICompensatableActivity` interface (i.e. the `CompensatableTransactionScopeActivity` and the `CompensatableSequenceActivity`)
Synchronization Activity

- **SynchronizationScopeActivity**
  - It is used to provide controlled access to variables or resources that are shared between parallel branches of execution.
  - It is a composite activity that manages the child activities that are added to it.
  - If two or more instances of the SynchronizationScopeActivity have the same synchronization handle, their execution is serialized so that the child activities of only one of them are executed at any one time.
Exceptions and Error Handling Activities

- **ThrowActivity**
  - It allows to throw a .NET exception declaratively as a workflow step

- **FaultHandlerActivity** and **FaultHandlersActivity**
  - They are used to catch and handle .NET exceptions within the workflow

- **CancellationHandlerActivity**
  - It is a container for cleanup logic associated with a composite activity