# Windows Workﬂow Foundation

## 1. Introduction

A workfow, then, is a series of steps to fnish a task. A workfow is often long

running and stateful, and often needs to wait on events and interact with humans.

We'll need to provide a mechanism to save the workfow's state of execution to a persistent data store and remove the running  workfow instance from memory. When a signifcant event occurs, we'll need to restore the workfow state and resume execution.

The workflow engine (runtime) will include all the required features like exception handling, tracking, and enabling cancellations.

Let's think about what we want in a workfow solution. We want to specify workfows in a declarative manner, perhaps with the aid of a visual designer. We want to feed workfow defnitions into a workfow engine. The engine will manage errors, events, tracking, activation, and de-activation. That what WWF and visual studio can do!

### **Activities**

The primary building block in Windows Workfow is the activity. Activities compose the steps, or tasks in a workfow, and defne the workfow. We can arrange activities into a hierarchy and feed activities to the workfow engine as instructions to execute. The activities can direct workfows involving both software and humans.

WF ships with a set of ready-made activities in the base activity library. The primitive activities in the library provide a foundation to build upon, and include control‑fow operations, like the IfElseActivity and the WhileActivity. The base activity library also includes activities to wait for events, to invoke web services, to execute a rules engine, and more.

$

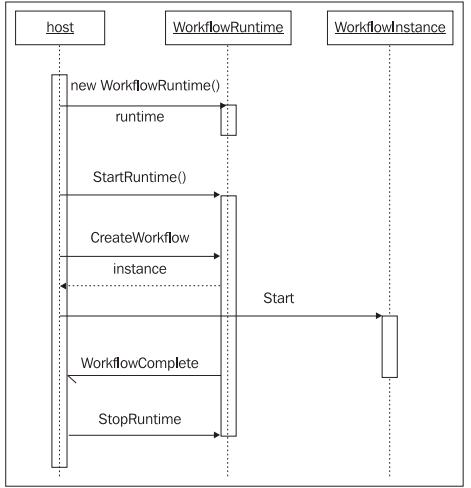
Windows Workfow allows developers to extend the functionality of the base activity

library by creating custom activities to solve problems in their specifc domain.

### **Windows Workfow Runtime**

 In Windows Workfow, the processor is in the WF runtime. We also need a host for the runtime.

Windows Workfow is not a stand‑alone application. Like ASP.NET, WF lives inside a handful of assemblies (most notably for this topic, the System.Workflow.Runtime.dll assembly).We can host WF in IIS (typically) a smart client application, a console application, or a Windows service, for instance.



Creating an instance of the WorkflowRuntime class and calling StartRuntime is all we need to spin up the workfow execution environment.

### Runtime Services

AddService allows us to make one or more services available to the runtime.  These services might be custom services we've written specifcally for our domain, like a custom scheduling service, or they might be services already written by Microsoft and included with WF.

#### Scheduling Service

A scheduling service controls threads the runtime needs to execute workfows, WF delivers DefaultWorkflowSchedulerService and  ManualWorkflowSchedulerService,

#### Transaction Services

A transaction service, allows the runtime to keep the internal state of a workfow consistent with the state in a durable store, like a relational database. The default transactional service is an instance of the DefaultWorkflowTransactionService

#### Persistence Services

A persistence service is responsible for saving the state of a workfow to a durable store. The SqlWorkflowPersistenceService service saves the state of a workfow into a SQL Server database. Persistence is required for long‑running workfows.

#### Tracking Services

A tracking service is responsible for monitoring and recording information about the execution of a workfow. WF includes a SqlTrackingService class that stores tracking data into a SQL Server

database.

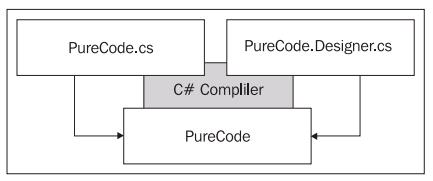
### Summary

The instructions for the WF engine are activities, and we can arrange these activities using a graphical designer, XAML, code, or a combination of the three. WF provides the services we need for a workfow engine, including persistence, threading, and transaction services. The future looks bright for building workfow solutions.

## Authoring Workfows

### Pure Code

Building a workfow with a pure code approach means we are only using C# or Visual Basic code to defne the workfow. There is no XAML involved. The code residents in  a PureCode.cs fle and a PureCode.Designer.cs fle



A workfow ultimately becomes a group of managed objects in memory. The trick is to arrange the objects in a relationship so they can perform useful work.

### Pure XAML

<SequentialWorkflowActivity

  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"

  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

  x:Class="MyWorkflow"

  >

  <CodeActivity ExecuteCode="SayHello" />

  <x:Code>

    <![CDATA[

    private void SayHello(object sender, EventArgs e)

    {

      Console.WriteLine("Hello, workflow!");

    }

    ]]>

  </x:Code>

</SequentialWorkflowActivity>

### Using Custom Activities in XAML

<SequentialWorkflowActivity

  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"

  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

  xmlns:otc="http://schemas.OdeToCode.com/WinWF/Activities"

  x:Class="MyWorkflow"

  >

  <otc:WriteLineActivity Message="Hello, workflow!"/>

</SequentialWorkflowActivity>

In the assembly definition,

[assembly: XmlnsDefinition(

      "http://schemas.OdeToCode.com/WinWF/Activities",

      "OdeToCode.WinWF.Activities")

]

An alternative approach to namespace mapping is to embed the CLR namespace and assembly name directly in the XAML. Assuming our custom activity is inside an assembly by the name of Foo.dll, this approach would look like the following: xmlns:otc="clr-namespace:OdeToCode.

WinWF.Activities;assembly=Foo".

### Compiling Workfows

Windows Workfow provides two compilers for us to use. The frst compiler is the class WorkflowCompiler in the System.Workflow.ComponentModel.Compiler namespace. The second compiler is a command-line compiler.

The frst step is to validate every activity in the workfow defnition has a validation logic;After validation, the compiler will generate code (the default language is C#) into a temporary directory. The generated source code then serves as input to the C# or Visual Basic.NET compiler for compilation into an assembly.

using (AutoResetEvent waitHandle = new AutoResetEvent(false))

{

  runtime.WorkflowCompleted += delegate { waitHandle.Set(); };

  runtime.WorkflowTerminated += delegate { waitHandle.Set(); };

  Type workflowType = Type.GetType("MyWorkflow, purexaml3");

  WorkflowInstance instance = runtime.CreateWorkflow(workflowType);

  instance.Start();

  waitHandle.WaitOne();

}

Remember the workfow executes asynchronously on a thread from the Common Language Runtime (CLR) thread pool.

We can use the WorkflowCompiler and WorkflowCompilerResults classes to programmatically compile workfow defnitions and retrieve a new assembly.

WorkflowCompiler compiler = new WorkflowCompiler();

WorkflowCompilerParameters parameters;

parameters = new WorkflowCompilerParameters();

parameters.GenerateInMemory = true;

parameters.ReferencedAssemblies.Add("chapter2\_Host.exe");

string[] xomlFiles = { @"..\..\purexaml\purexaml3.xoml" };

WorkflowCompilerResults compilerResults;

compilerResults = compiler.Compile(parameters, xomlFiles);

workflowType = compilerResults.CompiledAssembly.GetType("MyWorkflow");

WorkflowInstance instance = runtime.CreateWorkflow(workflowType);

instance.Start();

* First, the WorkflowCompiler creates a new application domain on each call to the Compile method.
* Second, the Compile method will automatically load the new assembly into the current AppDomain if the GenerateInMemory parameter fag is set to true and the compilation is successful

### XAML Activation

In some scenarios, compilation can become a burden. Imagine a database containing a thousand or more workfow defnitions tailored and updated for specifc users. In this scenario, we might want to avoid the churn of creating new assemblies.

   TypeProvider typeProvider = new TypeProvider(runtime);

   typeProvider.AddAssembly(Assembly.GetExecutingAssembly());

   runtime.AddService(typeProvider);

   XmlReader reader = XmlReader.Create(@"..\..\purexaml\purexaml5.

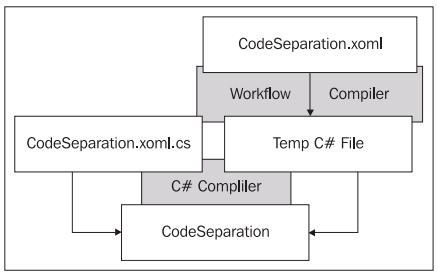
xoml");

   WorkflowInstance instance = runtime.CreateWorkflow(reader);

Activation is only available for workfows defned entirely in workfow mark‑up.

### Code and XAML Together

The second option uses XAML mark-up and code separation, also commonly referred to as code-beside.



### Summary

Code‑based workfow defnitions are perfectly reasonable to use for general‑purpose, fxed workfows.

XAML‑based workfow defnitions open up a number of additional possibilities. If we have a workfow defnition entirely in XAML we can use workfow activation and avoid compilation. Activation is useful when building a large number of dynamic workfows.

Using XAML with code‑beside fles will keep our workfow defnition in an XML fle. If we need to write custom tools for our workfows, like a custom workfow designer, then XAML is the preferred approach.

## Sequential Workflows

Windows Workfow offers two workfow execution styles out of the box: sequential and event-driven. A sequential workfow completes one activity and moves to the next, executing a sequence of consecutive steps.

Event‑driven workfows, on the other hand, rely on external events to drive them to a fnishing point. Event‑driven workfows model a workfow as a state machine.

### The SequenceActivity

A sequential workfow itself is an activity—a SequentialWorkflowActivity to be precise.

The SequentialWorkflowActivity class derives from the SequenceActivity class, which in turn derives from the CompositeActivity class.

The SequenceActivity class provides the logic to execute child activities. The SequenceActivity iterates through its children in a forward-only direction, executing each child once and then moving to the next child.

### Sequences Inside Sequences

The WhileActivity allows us to execute a single child activity until a condition returns false. In the Properties window for the WhileActivity, we can set Condition to Code Condition.

### Workfows and the Outside World

The basic mechanisms for communicating with a workfow include events, methods, and workfow parameters.

### Workfow Instance Lifetime Events

*WorkfowAborted*

*WorkfowCompleted*

*WorkfowCreated*

*WorkfowIdled*

*WorkfowLoaded*

*WorkfowPersisted*

*WorkfowSuspended*

*WorkfowResumed*

*WorkfowStarted*

*WorkfowTerminated*

*WorkfowUnloaded*

Workfow instance events are not the only technique available for monitoring the execution of a workfow. A workfow tracking service can receive exceptionally granular information about the state of a workfow.

### Workfow Parameters

An overloaded version of the runtime's CreateWorkflow method allows us to pass parameters into a new workfow instance. Parameters are a collection of name and value pairs in an instance of the Dictionary generic class.

Dictionary<string, object> parameters = new Dictionary<string,

object>();

parameters.Add("FirstName", "Scott");

parameters.Add("LastName", "Allen");

instance = runtime.CreateWorkflow(typeof(WorkflowParameters),

parameters);

When the workfow runtime creates a new workfow instance, it tries to fnd a companion property for each named parameter value. A companion property is a public, writable property on the workfow object with the same name as the parameter.

When a workfow completes, the runtime raises a WorkflowCompleted event and passes along a WorkflowCompletedEventArgs object. WorkflowCompletedEventArgs contains an OutputParameters property, which is a Dictionary collection of all output parameters.

### Raising Events and Invoking Methods

#### Service Contracts

A workfow and its host can exchange data via a local communication service (LCS). An LCS allows events and method calls between a workfow and a host. Behind the scenes, the workfow runtime works with an LCS to intercept communications and provide additional services.

**Events pass data from the host to a workfow, while methods pass data from a workfow to the host.**

[ExternalDataExchange]

interface IBugFlowService

{

    void AssignBug(Bug bug);

    event EventHandler<BugAddedArgs> BugAdded;

}

The workfow can invoke the AssignBug method to pass a Bug object to its host. Likewise, the host can raise the BugAdded event and pass data into a workfow via an event argument.

All objects passing between a workfow and a host must be serializable objects. If there is event,  we will need a serializable event argument class derives from ExternalDataEventArgs class

1.  An Iinterface, which defnes the communications allowed between host and workfows.

2.  A class, which holds the data we will pass back and forth.

3.  A class, which derives from ExternalDataEventArgs and carries data to the workfow during the BugAdded event.

#### Workfow Implementation

The two key activities from the base activity library are the HandleExternalEventActivity and the CallExternalMethodActivity.

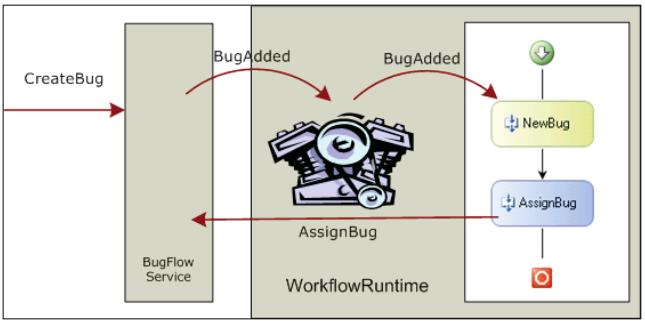
1. The host creates an ExternalDataExchangeService and adds the service to the workfow runtime. This service, provided by Windows Workfow, facilitates and manages all local communication services inside the workfow runtime, and will serve as the container for our own BugFlowService.

2.  The host creates an instance of the BugFlowService and adds the service to the list of services managed by the data exchange service. The data exchange service will fnd the IBugFlowService interface, and prepare to handle BugAdded events and AssignBug method calls.

3.  The host creates an instance of the BugFlow workfow, and starts the instance running. The workfow instance will wait for a BugAdded event.

4.  The host creates a new Bug object, and passes the bug to the bugFlow's CreateBug method.

At this point, we will turn to the visual depiction of the proceedings. The CreateBug method in our service will raise the BugAdded event. The workfow runtime (with help from the data exchange service) will catch the event, perform some processing, and pass the event to the workfow instance. The workfow instance might have been waiting for minutes, hours, days, or even months for this event to arrive, so we can't fre an event directly to the workfow instance. The workfow might not even be in memory at the time we raise the event; the runtime might have persisted the workfow into the database for long‑term storage. By intercepting the event, the workfow runtime has the chance to load the workfow into memory before passing the event along. When the workfow calls the AssignBug method, it takes a direct route to our local service, although workfow does have the opportunity to perform some pre- and post-processing on the method call.



### Faults

We can also intentionally raise an exception with a ThrowActivity.

Managing exceptions inside a workfow is similar to managing exceptions in Visual Basic or C#. Composite activities can include fault handlers to catch exceptions. If an activity does not andle an exception that occurs, the runtime will let the exception propagate to the parent activity.

The FaultHandlerActivity handles exceptions in Windows Workfow

We will associate each FaultHandlerActivity with a .NET exception type, such as System.NullReferenceException or System.ArgumentException.

Just like exception handlers in a general-purpose programming language,the purpose of a fault handler is to clean up or reverse any partially completed work. The workfow runtime will raise the WorkflowCompleted event if the SequentialWorkflowActivity fault handler handles an exception, as opposed to raising the WorkflowTerminated event if the exception goes unhandled.

## The Base Activity Library

There are activities for control fow, transaction management, local communication, web services, and more.

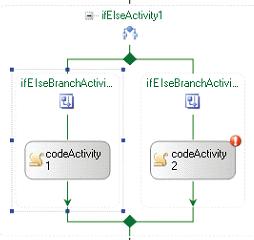
### The Basics

#### The CodeActivity

The Code activity's only interesting feature is its ExecuteCode event. in reality the Code activity should appear relatively infrequently and as a special case. Instead of using Code activities, we should look to package code into custom activities

#### The IfElseActivity

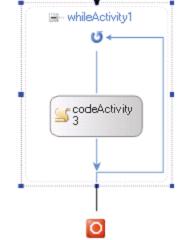
The IfElse activity is similar to the If…Then…Else statement in Visual Basic, and the if‑else in C#. Inside an IfElse activity are one or more IfElseBranch activities.



The Condition property of each branch can be confgured as a declarative rule (which the designer persists to an external .rules fle in XML format), or as a code condition (an event handler).

#### The WhileActivity

he WhileActivity can hold only a single child activity inside.Yet it can be a composite activity.



#### The SequenceActivity

 The activities inside a sequence execute one at a time, until the last activity completes.

#### The SuspendActivity

A host can subscribe to the workfow runtime's WorkflowSuspended event

#### The TerminateActivity

a host cannot resume a terminated workfow.

#### The ThrowActivity

 using a Throw activity makes the exception an explicit piece of the workfow model

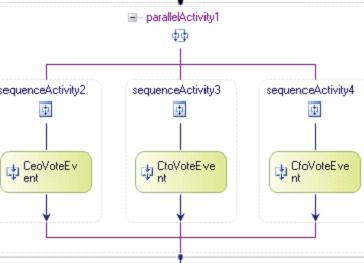
#### The InvokeWorkfowActivity

The InvokeWorkflow activity will asynchronously execute another workfow. Since the execution is asynchronous, we cannot retrieve output parameters from the other workfow,this activity will fre an Invoking

event.

#### The ParallelActivity

The Parallel activity allows multiple activities to execute at the same time. Note:only a single thread will execute inside a workfow.



#### The DelayActivity

The Delay activity will initialize a timer and wait for the timer to expire.

#### The ListenActivity

Like the Parallel activity, the Listen activity can contain multiple branches. Unlike the Parallel activity, the goal of a Listen activity is to fnish just one branch. The branches of a Listen activity are EventDriven activities, and must start by the branches by waiting for an event (the frst child must implement the IEventActivity interface).

we can use a Delay activity inside a Listen activity to simulate a timeout.

#### The EventHandlingScopeActivity

SynchronizationScope activity can serialize access to shared resources, even across workfow instances

#### The Replicator Activity

The Replicator can process a collection of data either sequentially or in parallel, depending on the setting of the ExecutionType property.

The InitialChildData property will hold the list of data objects for the Replicator to process. The Replicator will create a clone of its child activity to process each item in the child data ollection. The Replicator will not fnish execution until all the children have fnished; however, there is an UntilCondition property that the Replicator will evaluate before starting, and after completion of each child.

### Local Communication Events

#### The CallExternalMethodActivity

The CallExternalMethod activity invokes a method on a local service.

The CallExternalMethod activity includes a MethodInvoking event. The event will fre just before the activity calls the external method, and gives us an opportunity to set up the parameters.

#### The HandleExternalEventActivity

HandleExternalEvent is a blocking activity, meaning the activity is not going to complete until the event arrives from a local service. If there is a chance the event will never arrive, or if the event needs to arrive within a span of time, then it's best to use this activity inside a ListenActivity

#### The Activity Generator

When wca.exe finds such an interface it will generate dedicated custom activities for calling the methods and handling the events of the interface.

### Fault Handling

#### The FaultHandlersActivity

Many composite activities (like the WhileActivity, ListenActivity, SequenceActivity, TransactionScopeActivity, and others) can handle faults from their child activities using a fault handlers view.

#### The FaultHandlerActivity

A FaultHandler activity is analogous to a catch statement in C# or Visual Basic. A FaultHandler can trap an exception and perform processing.The FaultHandlerActivity has a FaultType property. This property represents the type of exception we want to catch.

### Transactions and Compensation

#### The TransactionScopeActivity

the TransactionScope activity will start a transaction and implicitly enlist any activities it contains into the transaction. The TransactionOptions property controls the timeout and the isolation level of the transaction

#### Compensation

In WF, we can only formally compensate for activities that implement the ICompensatableActivity interface. The CompensatableSequenceActivity and the CompensatableTransactionScopeActivity are the two activities in the base class library that implement this interface.

#### The CompensatableSequenceActivity

A CompensatableSequence activity functions just like a Sequence activity with the

addition of a compensation handler.

#### The CompensatableTransactionScopeActivity

The CompensatableTransactionScope activity functions just like a TransactionScope activity but with the addition of a compensation handler.

#### The CompensateActivity

The Compensate activity starts the compensation of a previously completed and compensatable activity. We can only compensate for the activities that implement the ICompensatableActivity interface. Instead of event handler, we use an activity here.

### Conditions and Rules

#### The ConditionedActivityGroup

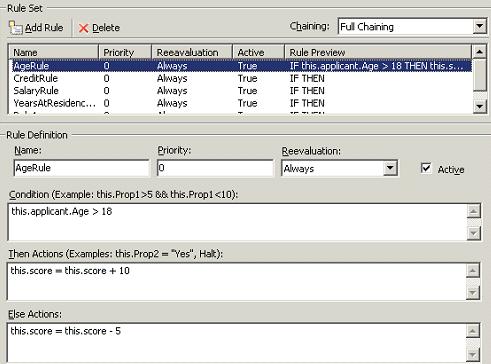
The CAG is a powerful activity that can use a combination of rules and code to reach a goal. The CAG conditionally executes activities until a condition evaluates to true. The CAG associates a WhenCondition with each activity in its storyboard, and the CAG will only execute an activity

if the activity's WhenCondition evaluates to true.

If we do not specify a WhenCondition for an activity, the activity will execute only once. If we do not specify an UntilCondition for the CAG, the CAG will continue to execute until all its activity's WhenCondition conditions return false.

#### The PolicyActivity

The Policy activity is a rules engine that allows us to separate business logic from the workfow and declaratively defne business policy. A Rule Set is a collection of rules for the Policy activity to execute, and each rule has conditions and actions.



### Web Services

#### The InvokeWebServiceActivity

#### The WebServiceInputActivity

#### The WebServiceOutputActivity

#### The WebServiceFaultActivity

### State Activities

Windows Workfow also supports state‑machine workfows. State machines are completely event driven. State machine workfows are a good ft for modeling a process where decisions come from outside the workfow.

### The StateActivity

The State activity represents one of the states in a state machine. Every state machine workfow needs an initial state and optionally a close state.Notice we can include a State activity inside a State activity. The recursive composition of states allows contained states to inherit the events and behavior of their containing state

#### The StateInitializationActivity

The StateInitialization activity is a sequence activity that contains child

activities. When the state machine transitions to a state, the children inside the

initialization activity will execute. We can only have one initialization activity per

state.

#### The StateFinalizationActivity

Like the StateInitializationActivity for finailization.

#### The EventDrivenActivity

The EventDriven activity is also a sequence activity with children inside. The EventDriven activity, however, only executes when an event arrives. The first child activity must implement the IEventActivity interface,

A State activity can contain more than one EventDriven activity inside.

#### The SetStateActivity

The SetState activity transitions a state machine to a new state.

## Custom Activities

Why Would I Build Custom Activities?

* Building reusable components
* Extending Windows Workfow
* Building a domain‑specifc language

### How Do I Build Custom Activities?

One approach uses composition and the second approach uses derivation.

The composition approach is a similar experience to authoring a workfow. We  use the designer to drag, drop, and confgure activities inside a new custom  activity, and then package the custom activity into an assembly for use in other workfow projects

We can customize the design view, validation, serialization, and code-generation

pieces of the activity. The derivation approach gives us the highest level of control

and offers a path to extending Windows Workfow with custom code.

### Activity Composition

A compiled composite activity becomes a black box. We cannot add, delete, or modify the child activities inside this black box. We need to use property promotion.

### Opening a Black Box

we can promote properties from inside the black box and expose the properties to the outside world.

#### Property Promotion

Property promotion establishes a connection from a property on a parent activity to a property on one of the parent's child activities. The property should be a DP.

### Derivation

In composition, we focus  on arranging child activities inside a custom activity, and we build bigger activities from smaller activities. With derivation, we focus on designing a single activity—its properties and execution model. The derivation approach is often the low-level approach

#### ConsoleWriteActivity

 Activity is the foundation for all activities in WF. By overriding the Execute method, we've taken full responsibility for the behavior of this activity.

public class ConsoleWriteActivity : Activity

    {

        public string Text

        {

            get { return \_text; }

            set { \_text = value; }

        }

        private string \_text;

        protected override ActivityExecutionStatus Execute(

            ActivityExecutionContext executionContext)

        {

            Console.WriteLine(Text);

            return ActivityExecutionStatus.Closed;

        }

    }

#### Activity Components

Two important component types are activity designers and activity validators.

#### Activity Validators

Activity validators execute during design and compilation to ensure our activity has the proper confguration for execution at run time.

public override ValidationErrorCollection Validate(

    ValidationManager manager, object obj)

{

    ValidationErrorCollection errors = base.Validate(manager, obj);

    ConsoleWriteActivity activity = obj as ConsoleWriteActivity;

    if (activity.Parent != null &&

       String.IsNullOrEmpty(activity.Text))

    {

        errors.Add(

              ValidationError.GetNotSetValidationError("Text"));

    }

    return errors;

}

[ActivityValidator(typeof(ConsoleWriteValidator))]

public class ConsoleWriteActivity : Activity

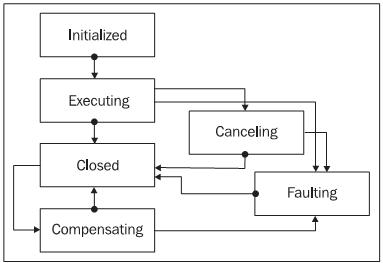
{}

#### Activity Designers

Activity designers control the appearance and behavior of an activity at design time.

#### Activity Execution

These states are represented by the ActivityExecutionStatus enumeration: Initialized, Executing, Closed, Canceling, Compensating, and Faulting.



#### Execution Context

An ActivityExecutionContext (AEC) epresents the execution environment of an activity. The AEC object is a gateway to the services inside the WF runtime (it provides a generic GetService method), and provides methods for scheduling the execution of activities (ExecuteActivity, CloseActivity, CancelActivity).

#### Custom Composite Activities

When we override the Execute method, we become responsible for managing the execution of our child activities. We have to manage this execution by coordinating with the WF runtime, and this coordination takes place using the AEC. Instead of calling Execute directly, we schedule execution of a child activity using the AEC's ExecuteActivity method.

protected override ActivityExecutionStatus Execute(

    ActivityExecutionContext executionContext)

{

    \_currentIndex = 0;

Activity child = EnabledActivities[0];

    child.Closed +=

        new EventHandler<ActivityExecutionStatusChangedEventArgs>

            (child\_Closed);

    executionContext.ExecuteActivity(child);

    return ActivityExecutionStatus.Executing;

}

void child\_Closed(object sender,

                  ActivityExecutionStatusChangedEventArgs e)

{

    ActivityExecutionContext context = sender

                             as ActivityExecutionContext;

    e.Activity.Closed -= child\_Closed;

    \_currentIndex++;

    if (\_currentIndex < EnabledActivities.Count)

    {

        Activity child = EnabledActivities[\_currentIndex];

        child.Closed +=

            new EventHandler<ActivityExecutionStatusChangedEventArgs>

                (child\_Closed);

        context.ExecuteActivity(child);

    }

    else

    {

        context.CloseActivity();

    }

}

In an context, each activity only reaches the Executing state once. If we need to execute an activity more than once we need to create a new execution context.  the While activity spawns a new ActivityExecutionContext each time it executes its child activity. Spawning an AEC creates a clone of the child activity. The original child activity will be known as the template activity.

ActivityExecutionContextManager manager;

 manager = executionContext.ExecutionContextManager;

 ActivityExecutionContext newContext;

 newContext = manager.CreateExecutionContext(EnabledActivities[0]);

 newContext.Activity.Closed +=

     new EventHandler<ActivityExecutionStatusChangedEventArgs>

         (Activity\_Closed);

 newContext.ExecuteActivity(newContext.Activity);

f the activity inside the While activity needs to perform compensation, it does so by forcing each cloned activity  to compensate its particular unit of work.

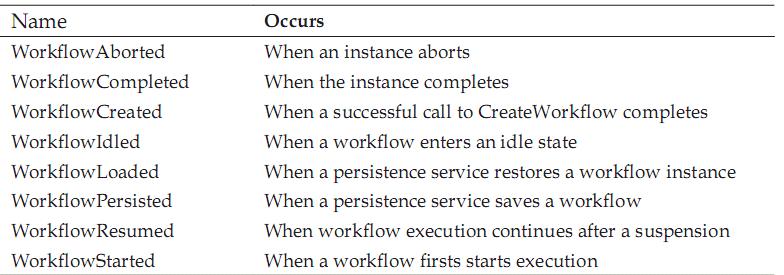
## Workfow Hosting

All the host processes needs is the ability to load the .NET 3.0 center. All the host processes needs is the ability to load the .NET 3.0 . All the host processes needs is the ability to load the .NET 3.0 Workfow assemblies

A host can also customize the workfow runtime by layering additional services on top of the runtime's base feature set. These services can provide persistence support for long‑running workfows, tracking support for monitoring workfow execution, and more.

### The Workfow Runtime

The WorkflowRuntime class is the host's gateway to Windows Workfow



using(WorkflowRuntime runtime = new WorkflowRuntime())

using(AutoResetEvent reset = new AutoResetEvent(false))

{

    runtime.WorkflowCompleted += delegate { reset.Set(); };

    runtime.WorkflowTerminated += delegate { reset.Set(); };

    runtime.StartRuntime();

    WorkflowInstance instance;

    instance = runtime.CreateWorkflow(typeof(SimpleWorkflow));

    instance.Start();

    reset.WaitOne();

}

 Most applications will keep the runtime around for the life of the process and run multiple workfows.

#### Workfow Runtime Logging

Windows Workfow uses this tracing API to log information about what is happening inside the runtime

We can't use a debugger to step into the code of the WF runtime, but we can enable logging to see what is happening inside.

The available logging values are Critical, Error, Warning, Information, and All.

 If we want to send logging information to another destination, we can create a new trace listener.

#### Workfow Runtime Confguration

* The declarative approach confgures services using the application's confguration fle.
* The imperative approach creates services in code and adds them to the runtime with the AddService method of the WorkflowRuntime class.

### Scheduling Services

Scheduling services in WF are responsible for arranging workfows onto threads for execution. DefaultWorkflowSchedulerService and the ManualWorkflowSchedulerService.

he default scheduling service will schedule workfows to run on threads from the process‑wide CLR thread pool. workfows execute asynchronously on a background thread by default.

A host using the manual scheduling service must donate threads to the workfow runtime.We can use the manual scheduling service to execute workfows synchronously

The manual service will use the calling thread to execute the workfow synchronously. This is

how a host donates a thread.

#### Scheduling Parameters

maxSimultaneousWorkflows

Most smart client applications will work well with the default scheduling service. . Server-side applications will generally want to use the manual scheduler and donate the request thread for workfow execution.

### Persistence Services

Persistence saves the state of the workfow into long‑term storage. When the event fnally arrives, the runtime can restore the workfow and resume processing.

The workfow runtime decides when to persist workfows, and the persistence service decides how and where to save the workfow state.

* When an atomic transaction inside a TransactionScope activity or CompensatableTransactionScopeActivity activity completes
* When the host application calls the Unload or RequestPersist methods on a WorkflowInstance object
* When a custom activity with the PersistOnClose attribute completes
* When a CompensatableSequence activity completes
* When a workfow terminates or completes

A persistence service can use the opportunity to clean up workfow state left behind from previous operations.

WPF has the SqlWorkflowPersistenceService implemented.

#### Persistence and Serialization

In order for the persistence service to save the state of a workfow, it frst has to serialize the workfow.

the runtime uses the BinaryFormatter class from the base class library when persisting workfows. The BinaryFormatter will attempt to serialize every piece of state information inside our workfow, including a custom object like our Bug object. If there is a feld that we don't need to serialize and restore with the workfow instance, we can tell the formatter to skip serialization with the NonSerialized attribute.

### Tracking Services

 tracking features to capture and record information about workfow execution.

Both logging and tracing expose detailed information about important events inside the workfow runtime. However, tracking information is exposed through an API specialized for Windows Workfow. Logging has no API and can be configured for different sources, e.g. Critical, Warning, like how we deal with warnings for compilers.

Tracking profle defnes and flters the type of information we want to receive from the runtime. A tracking channel is a communications conduit between the workfow runtime and the tracking service.

SqlTrackingService

WF provides an optimization for applications using both the SQL persistence and SQL tracking services with the SharedConnectionWorkflowCommitWorkBatchService class

### Summary

WF services include scheduling services (to manage threads), persistence services (to manage state), and tracking services (to record tracking information). If the built‑in services do not fulfl our requirements, we can replace any of them with custom versions we write ourselves.

## Event-Driven Workfows