

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 04 – Creating State Machine Workflows in C#

Information in this document, including URL and other Internet Web site references, is subject to change without notice. This document supports a preliminary release of software that may be changed substantially prior to final commercial release, and is the proprietary information of Microsoft Corporation.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

The entire risk of the use or the results from the use of this document remains with the user. Complying with all applicable copyright laws is the responsibility of the user.  Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document.  Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2008 Microsoft Corporation.  All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, Windows Server, Visual C# and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

**Contents**

[Overview 1](#_Toc219631153)

[Objectives 1](#_Toc219631154)

[System Requirements 1](#_Toc219631155)

[Setup 1](#_Toc219631156)

[Physical Folder Structure 2](#_Toc219631157)

[Code Snippets 2](#_Toc219631158)

[Starting Material 2](#_Toc219631159)

[Acronyms Used in this Lab 2](#_Toc219631160)

[Scenario 2](#_Toc219631161)

[Exercise 1 – Create the Order State Machine Workflow 3](#_Toc219631162)

[Task 1 – Create a new State Machine Workflow Project 4](#_Toc219631163)

[Task 2 – Define the States for the Order State Machine Workflow 8](#_Toc219631164)

[Task 3 – Define Workflow variables 12](#_Toc219631165)

[Task 4 – Define the WaitingForOrderState 14](#_Toc219631166)

[Task 4 Summary 23](#_Toc219631167)

[Task 5 – Define the OrderOpenState 23](#_Toc219631168)

[Task 6 – Define the OrderProcessedState 31](#_Toc219631169)

[Task 7 – Add the OrderApplication Project 38](#_Toc219631170)

[Task 8 – Test the State Machine Workflow 39](#_Toc219631171)

[Exercise 2 – Use the StateMachineTracking Service 42](#_Toc219631172)

[Task 1 – Add a reference to the StateMachineTracking Service 42](#_Toc219631173)

[Task 2 – Modify the code in the OrderApplication 43](#_Toc219631174)

[Task 3 – Test the OrderApplication and State Machine Tracking 50](#_Toc219631175)

[Lab Summary 52](#_Toc219631176)

# Overview

Estimated time to complete this lab: **60 minutes**

This lab introduces another workflow authoring style known as State Machine Workflows.

**State Machine Workflows** offer a flexible workflow creation style where the process is modeled as a state machine. Unlike a **Sequential Workflow** where the activities execute in a sequence, in a **State Machine Workflow** the activities execute based on external events. The state machine responds to an external event, does the required work and then moves to the next state.

## Objectives

After completing this lab, you will be able to:

* Author **State Machine Workflows** using the **Visual Studio 2008** designer for **Windows Workflow Foundation**, referred to as the Visual Studio workflow designer in the remainder of this document.
* Utilize the **State** and **SetState** activities.
* Utilize the sample state machine tracking service to inspect and track the state of a workflow from the host application.

More information about Windows Workflow Foundation can be found at <http://msdn.microsoft.com/workflow>

## System Requirements

* Microsoft Visual Studio 2008

## Setup

Unzip the lab to your local hard drive. Everything needed to complete the lab is in the zip file.

### 

### Physical Folder Structure

File paths referenced in this lab assume the lab is installed in the following folder:

*C:\WF\WF 3.5 Labs\Lab[Number]*

Within the ***Lab[Number]*** folder, several child folders are available:

* ***CSharp*** – The lab written for C#
* ***VB*** – The lab written for VB
* ***resources*** –Any files referenced in the lab can be found in the Resources subdirectory, including source code for custom assemblies referenced in the exercises.

Within each *[Language]* folder, several child folders are available:

* ***before*** – The work area for completing the HOL
* ***after*** – The fully completed HOL

### Code Snippets

All code required for this lab consisting of more than 2 lines is available as code snippets. To learn more about code snippets including how to install them and how to use them, see the snippet guide document for the language of your choice in the folder:

*C:\WF\WF 3.5 Labs\Snippets*

## Starting Material

### Acronyms Used in this Lab

* WF – Windows Workflow Foundation

### Scenario

The scenario presented in the lab is that of an order processing application, potentially usable by a company taking orders for merchandise to be shipped. The orders are entered, updated with revisions before shipment, and finally shipped to customers.

# Exercise 1 – Create the Order State Machine Workflow

In this exercise, you will create a **State Machine Workflow** for managing an order. You can imagine an e-commerce application where an order might go through several states before it is completed. For lab purposes, consider the following four states:

* **WaitingForOrderState**
* **OrderOpenState**
* **OrderProcessedState**
* **OrderCompletedState**

In the order system rules specify which events may occur depending on an order’s current state. For example:

* An order that is open (is in the **OrderOpenState** state) can be updated, processed, canceled, or shipped.
* An order that is processed (is in the **OrderProcessedState** state) can be updated or shipped, but cannot be canceled.

As an event occurs, the State Machine Workflow can optionally transition the state for an order. For example:

* When an order is open and the **OrderShipped** event occurs, the **State Machine Workflow** transitions the completed state for the order.

The states and transitions for the simple order State Machine Workflow are illustrated in Figure 1 below.

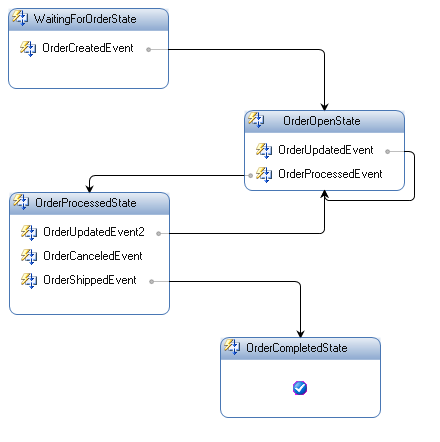


Figure : States in the State Machine Workflow

We will start creating the Order State Machine workflow by modeling the possible states for an order entity by using the **State** activity. Then we will specify the events that can occur with each state by using **EventDriven** activities and custom activities for the order events.

In most cases when an order event occurs we will transition the state of the order using the **SetState** activity. Finally, after authoring our workflow we will test it using a pre-built Windows Forms host.

## Task 1 – Create a new State Machine Workflow Project

#### Creating the Visual Studio solution

1. Open Visual Studio 2008 by going to the   
   **Start Menu | All Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**
2. Select the **File | New | Project** menu command.

The **New Project** dialog window appears.

1. In the **New Project** dialog window, expand **Visual C# | Workflow** in the Project Types tree on the left side.
2. Select the template named **State Machine Workflow Library** and enter the following values for the name and location:

**Name:** *OrderWorkflows*

**Location:** *C:\WF\WF 3.5 Labs\Lab04\CSharp\before*

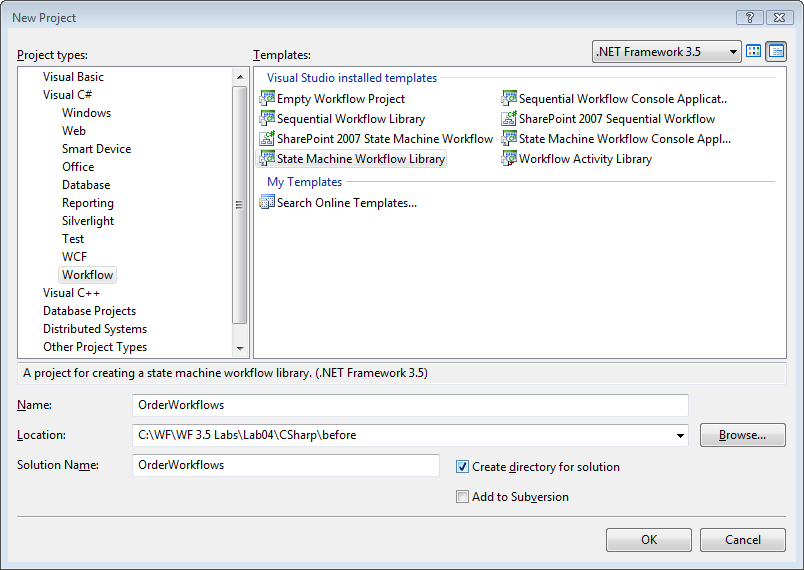


Figure : New Project Dialog

1. Click **OK**

You should now have a new solution and workflow project.

Note also that under the project *OrderWorkflows* there is a file called *Workflow1.cs* added as sown in Figure 2.

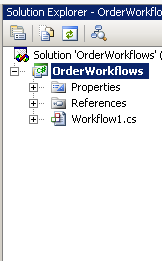


Figure : The project in Solution Explorer

Also you can see that the state machine designer is displayed.

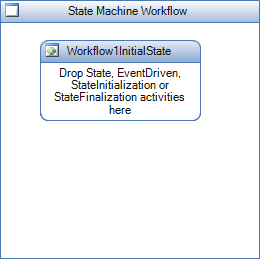


Figure : The State Machine Designer

#### Setting up project references

The project needs a reference to the order services that will provide the events the state machine workflow uses.

1. Right-click on the *OrderWorkflows* project in Solution Explorer and select **Add Reference.**

The **Add Reference** dialog appears

1. Select the **Browse** tab and browse to the following directory:

*C:\WF\WF 3.5 Labs\Lab04\resources\bin*

1. Select the*OrderLocalServices.dll*file as shown in Figure 5.
2. Click **OK**.

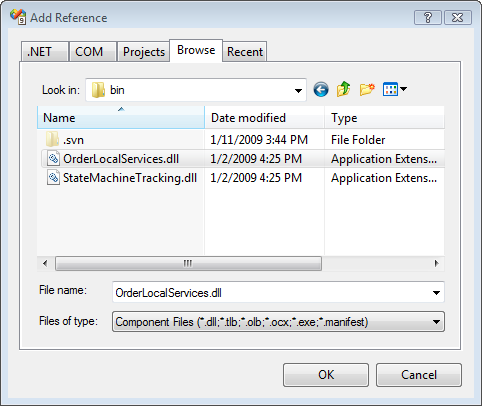


Figure : Add Reference Dialog

## Task 2 – Define the States for the Order State Machine Workflow

We are now ready to define the workflow using the **State Machine** designer.

#### Defining the first workflow state

1. Double click on the **Workflow1.cs** file.   
   The **Visual Studio workflow designer** for the **State Machine Workflow** should be displayed.

|  |
| --- |
| The State Machine Workflow Designer The designer for the State Machine Workflow is a free form designer. You can move the state activity around and place it in any position you want on the drawing surface. As the state machine is built, the transitions will also be drawn. Select the *Workflow1InitialState*in the diagramand try moving it around. You can drag it anywhere on the surface. |

1. Select the **Workflow1InitialState** activity in the Visual Studio workflow designer.
2. Rename **Workflow1InitialState** activity to ***WaitingForOrderState.***

In the properties window, change the **(Name)** property*.*

The Visual Studio workflow designer now shows a red exclamation point indicating there is an error with the workflow.

1. Click on the exclamation point and select the error from the smart tag as shown in Figure 6:

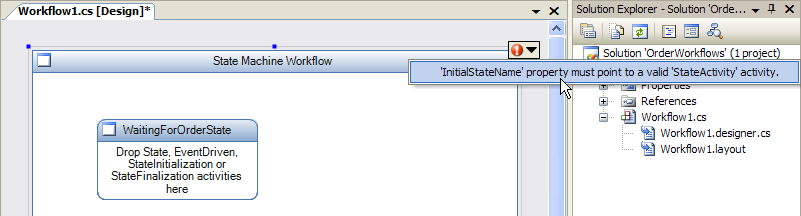


Figure : Workflow error

1. Change the **InitialStateName** property to **WaitingForOrderState**.

This specific that the initial state for the workflow is the state just defined.

#### Adding more states to the workflow

1. In the Visual Studio workflow designer, select the **View | Toolbox** menu command to display the toolbox with available activities.
2. Add a **State** activity to the workflow by dragging and dropping it out of the toolbox.
3. In the properties window, change the **(Name)** property for the new State activity to **OrderOpenState**.

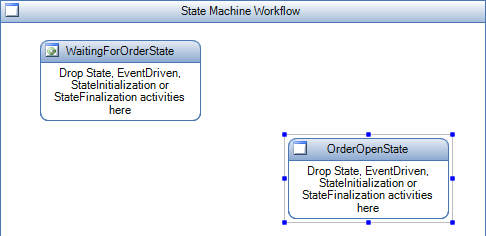


Figure : New State Activity in the designer

1. Add a **State** activity to the workflow definition by dragging and dropping it out of the toolbox.
2. In the properties window, change the **(Name)** property for the new **State** activity to **OrderProcessedState**.

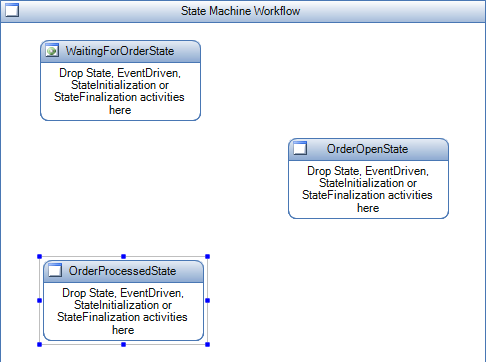


Figure : Current workflow view

1. Add a **State** activity to the workflow definition by dragging and dropping it out of the toolbox.
2. In the properties window, change the **(Name)** property for the new State activity to **OrderCompletedState**.

Your workflow definition should look like Figure 9:

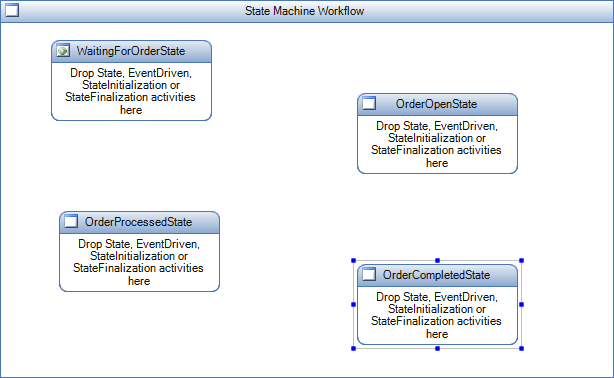


Figure : Current workflow view

1. Select the state machine workflow in the Visual Studio workflow designer by clicking on any empty space on the drawing surface.
2. In the properties window, change the **CompletedStateName** property for the State Machine Workflow to **OrderCompletedState**.

An icon appears on the **OrderCompletedStatus** state indicating it as the state the workflow identifies as the completed state. See Figure 10.

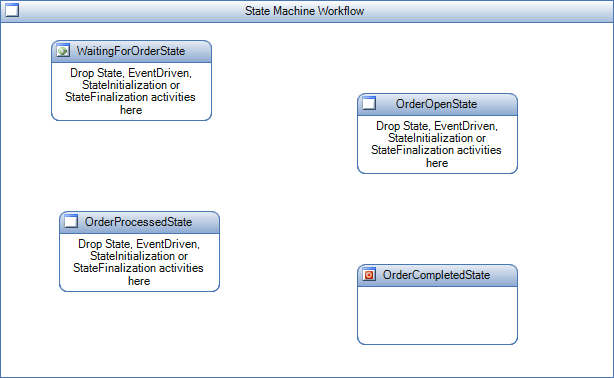


Figure : Current workflow view

## Task 3 – Define Workflow variables

Before configuring properties of the workflow and states, add a variable to which properties of an activity in the workflow may be bound.

#### Adding a bind-able workflow property

1. Right click on the file **Workflow1.cs** and select **View Code**.
2. Add the following code within the class **Workflow1.**

Newly added code in shown highlighted in yellow.

|  |
| --- |
| **Snippet:** WFLab04\_Ex01\_Task3\_Variable |
| **public sealed partial class Workflow1 : StateMachineWorkflowActivity**  **{**  **public Workflow1()**  **{**  **InitializeComponent();**  **}**  **private string orderCanceledError = "Order has been canceled";**  **public string OrderCanceledError**  **{**  **get { return orderCanceledError; }**  **}**  **}** |

## Task 4 – Define the WaitingForOrderState

|  |
| --- |
| The EventDriven Activity It is a **CompositeActivity** that is used to handle an event; typically it can be raised from the host or by the runtime in response to a delay timer expiring. **EventDrivenActivity** is inherited from **SequenceActivity**; therefore, it is a sequence that has the additional restriction that the first activity should be an **IEventActivity**. |

#### Adding an activity to a state

1. Right click **Workflow1.cs** and select **View Designer**.
2. Drag an **EventDriven** activity from the toolbox to the **WaitingForOrderState**.
3. In the properties window, change the **(Name)** property for the new **EventDriven** activity to **OrderCreatedEvent**.

Your workflow should look like Figure 11:

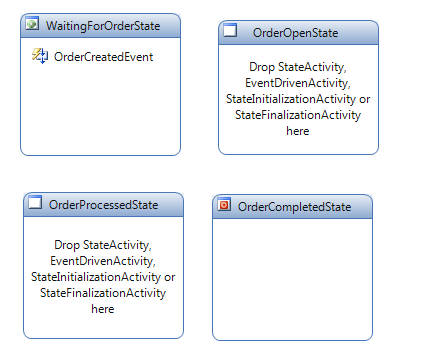


Figure : Current workflow view

1. Double click on the **OrderCreatedEvent** activity. The Visual Studio workflow designer will now show the contents of the **EventDriven** activity.

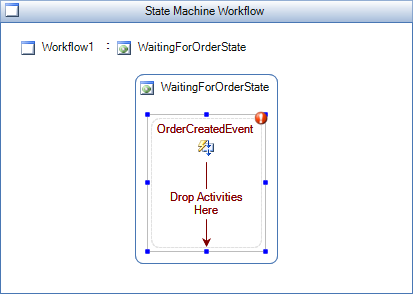


Figure : An activity within the state

|  |
| --- |
| The HandleExternalEventActivity Activity The HandleExternalEventActivity activity is used in conjunction with the **CallExternalMethodActivity** activity for input and output communications with a local service. You can use these activities directly for generic communications. Or, you can subclass the **HandleExternalEventActivity** and **CallExternalMethodActivity** classes to create activities that are strictly bound to specific events and methods on an interface that is attributed with the **ExternalDataExchangeAttribute** attribute. |

1. Drag a **HandleExternalEvent** activity to the workflow definition from the toolbox into the **OrderCreatedEvent** EventDriven activity as shown in Figure 13.

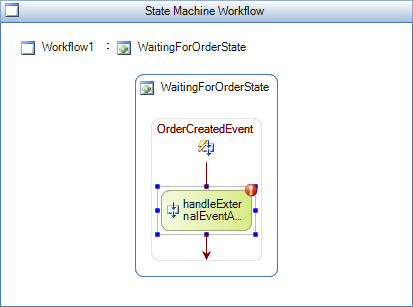


Figure : HandleExternalEvent activity

1. There is a validation error on the **HandleExternalEvent** activity. Configure the activity to ensure it can receive the **OrderCreated** event defined in the **IOrderService** interface.

Clicking on the red exclamation point reveals the **InterfaceType** and **EventName** properties need to be set as seen in Figure 14.

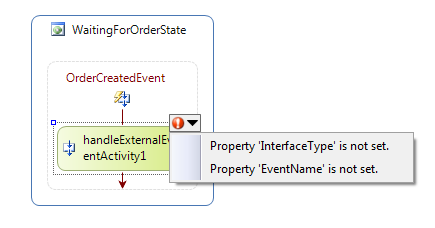


Figure : Properties not yet configured

#### Setting the InterfaceType property

1. Select the option **“Property ‘InterfaceType’ not set.**

This opens the **InterfaceType** property in the properties window for **handleExternalEventActivity1**.

1. Click on the ellipse button displayed next to the input box as shown in Figure 15.

The **browse and** **select a .NET Type** dialog is displayed.

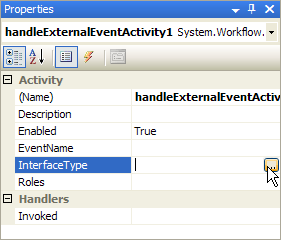


Figure : InterfaceType in the Properties dialog

1. Expand **Referenced Assemblies** and select the **OrderLocalServices** assembly.

The **IOrderService** interface will be displayed.

1. Select that the **IOrderService** interface and click **OK**.
2. Select **OrderCreated** from the drop down for the **EventName** property.

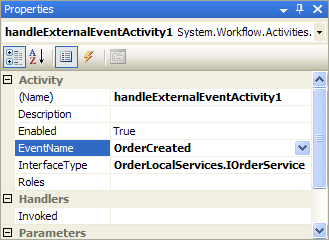


Figure : EventName in the Properties Dialog

You have now configured the **HandleExternalEvent** activity to receive the **OrderCreated** Event from the **IOrderService** interface.

When you select the **OrderCreated** event type, 2 more properties appear that must be configured, the **e** property and the **Sender** property.

1. Select the **e** property and click the ellipsis button next to it to bring up the activity binding interface.
2. Select the **Bind to a new member** tab as shown in Figure 17.

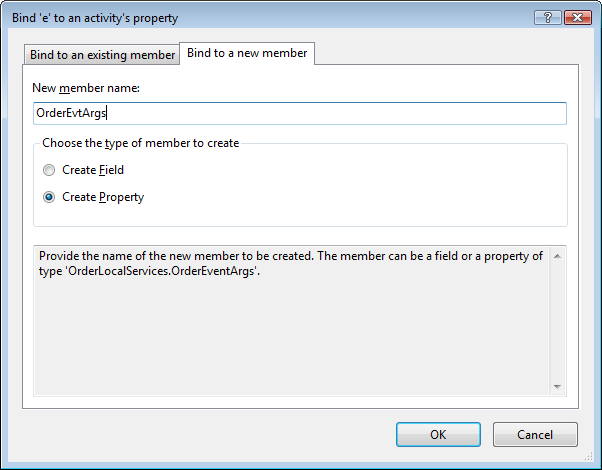


Figure : Activity Binding dialog

1. Enter **OrderEvtArgs** in the **New member name** box as shown in Figure 17.
2. Click **OK**.
3. For the **sender** property, click the ellipsis button and use the activity binding interface to bind **sender** to a new member variable named **OrderSender**.

The resulting properties window looks like Figure 18.

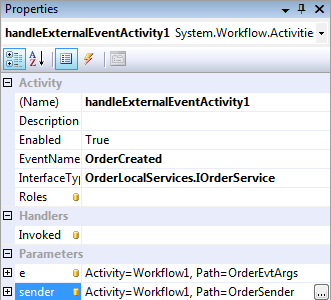


Figure : handleExternalEventActivity1 in the Properties Dialog

#### Adding a SetStateActivity

|  |
| --- |
| The SetState Activity A state machine by nature uses information that it receives through various events to move from one state to another. This transition between states is performed by the SetStateActivity activity. The target of a SetStateActivity activity must be a "leaf" state that does not contain any other states. |

1. Add a **SetState** activity to the workflow definition by dragging and dropping it out of the toolbox into the **OrderCreatedEvent** **EventDriven** activity, below the **HandleExternalEvent** activity.
2. With the new **SetState** activity selected, in the properties window change the **TargetStateName** property to **OrderOpenState**.

Your workflow definition should look like Figure 19.

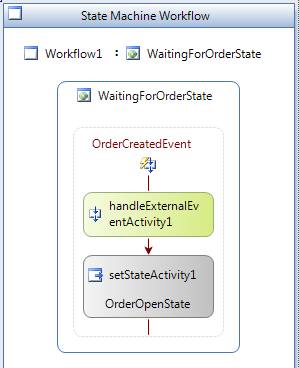


Figure : setStateActivity1

1. Navigate back to the **State view** in the **State Machine Workflow** by clicking on the **Workflow1** link in the **Visual Studio workflow designer**.

Alternatively, you can use the **Document Outline tool window** to view the entire workflow as a hierarchical tree. Open the **Document Outline window** by selecting the menu item:

**View | Other Windows | Document Outline**.

The workflow diagram now shows that association between states as a connecting arrow.

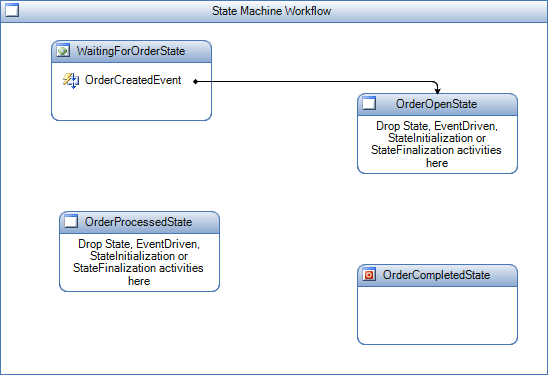


Figure : Current view of the workflow

### Task 4 Summary

The following steps summarize the work performed in Task 4. This is just a summary, don’t follow these steps.

1. Dropped an event driven activity inside the **State** activity.
2. Double clicked on the **EventDriven** activity to look at the details of the **EventDriven** activity.
3. Dropped a **HandleExternalEvent** activity inside the event driven activity.
4. Configured the **HandleExternalEvent** activity -
5. Configured the interface and the event that can be received.
6. Configured the fields on the event payload by binding them to workflow variables.
7. Defined a **SetState** activity and defined a transition to the next state.

## Task 5 – Define the OrderOpenState

|  |
| --- |
| Reusing the EventDriven activitiesghIn the following tasks we will be adding more event handlers and state transitions. All the events come from the same source and use the same arguments, so we are going to copy and paste the EventDriven activities and change some of the properties, rather than create them all over again. |

The configuration of the **OrderOpenState** and the **OrderProcessedState** is similar to the **WaitingForOrderState**. In task 5 you will configure the **OrderOpenState** to receive the following events:

* [**OrderUpdatedEvent**](#_Define_the_OrderUpdated_Event) – Transition back to the **OrderOpenState**.
* [**OrderProcessedEvent**](#_Define_the_OrderProcessed_Event) – Transition to the **OrderProcessedState**.

#### Defining the OrderUpdated Event

1. In the design view for the state machine workflow, right-click the **OrderCreatedEvent** from task 4 in the **WaitingForOrderState** and select **Copy**.
2. Select the **OrderOpenState**, right-click and select **Paste**.

This automatically takes you to the design view for the new **EventDriven** activity you pasted.

1. Change the **(Name)** property for **eventDrivenActivity1** to **OrderUpdatedEvent**.
2. Select the **HandleExternalEvent** activity and change the **EventName** property to **OrderUpdated**.
3. Set the property **e** to **Workflow1*.OrderEvtArgs*** by clicking the ellipsis button next to it and selecting the **OrderEvtArgs** member as shown in Figure 21.
4. Click **OK** to close the window and bind the property.

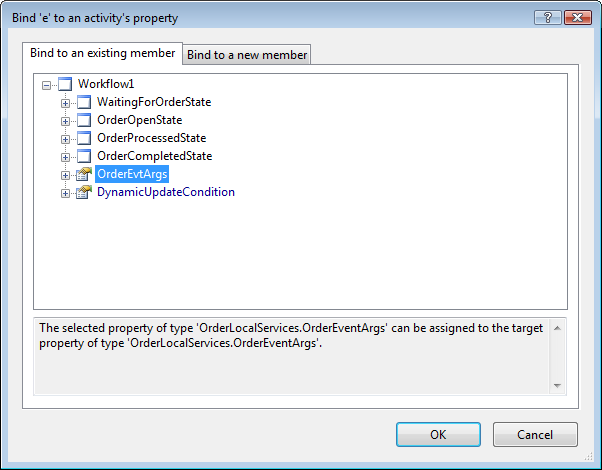


Figure : Activity Binding dialog

1. Click on the ellipsis button next to **sender** and use the activity binding interface to bind it to **Workflow1.OrderSender**.

The properties window should now look like Figure 22.

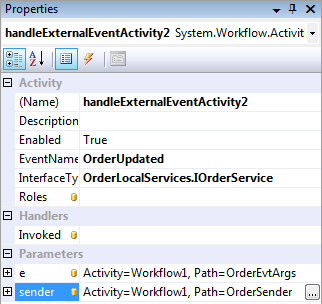


Figure : handleExternalEventActivity2 in the Properties Dialog

1. Leave the target state of the **SetState** activity set to **OrderOpenState**.
2. Navigate back to the **State view** in the **State Machine Workflow** by clicking on the **State Machine Workflow** link in the Visual Studio workflow designer.

Alternatively, you can use the **Document Outline tool window** to view the entire workflow as a hierarchical tree. Open the **Document Outline window** by selecting the menu item:

**View | Other Windows | Document Outline**

This shows the document outline view shown in Figure 23.

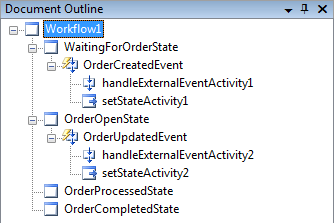


Figure : Document Outline View

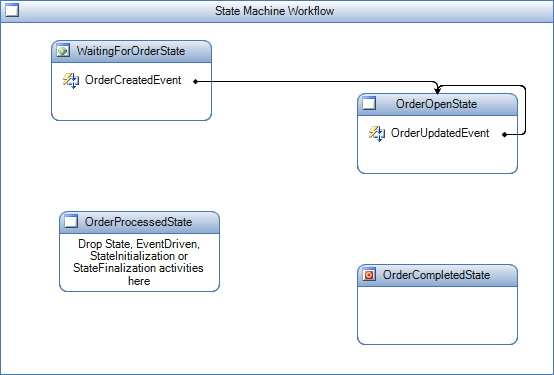


Figure : Current view of workflow

#### Adding a second activity to the OrderProcessed Event

1. In the design view for the state machine workflow, right-click the **OrderCreatedEvent** from task 4 in the **WaitingForOrderState** and select **Copy**.
2. Select the **OrderOpenState**, right-click and select **Paste**.

This automatically opens the design view for the new **EventDriven** activity.

1. Change the **(Name)** property for **eventDrivenActivity1** to **OrderProcessedEvent**.
2. Select the **HandleExternalEvent** activity and change the **EventName** property to **OrderProcessed**.
3. Set the property **e** to **Workflow1.OrderEvtArgs**.
4. Set the **sender** to **Workflow1.OrderSender** using the activity binding interface.

The properties window should now look like Figure 25.

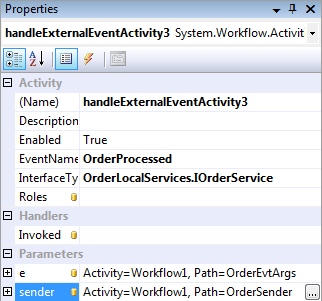


Figure : handleExternalEventActivity3 in the Properties Dialog

1. Navigate back to the **State view** in the **State Machine Workflow** by clicking on the **Workflow1** link in the Visual Studio workflow designer.

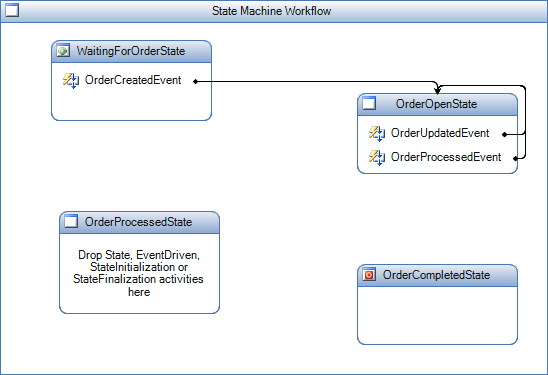


Figure : Current view of workflow

The order processed event currently sets the state to **OrderOpenState**. It must link to **OrderProcessedState** instead.

This could be done by manually setting the **TargetStateName** property on the **SetState** activity or by using drag and drop in the workflow designer.

1. Select the **OrderProcessedEvent** connector as shown below.

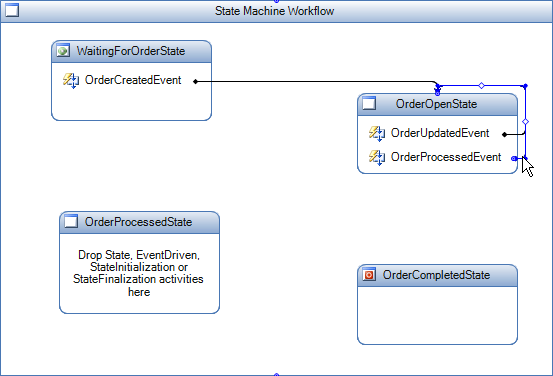


Figure : Current view of workflow

1. The connector currently terminates at the top of the **OrderOpenState**.

Move the mouse cursor over the point where it terminates and the cursor will change to a square target.

1. Click and hold to drag the connector to the top of the **OrderProcessedState** as shown in Figure 28.

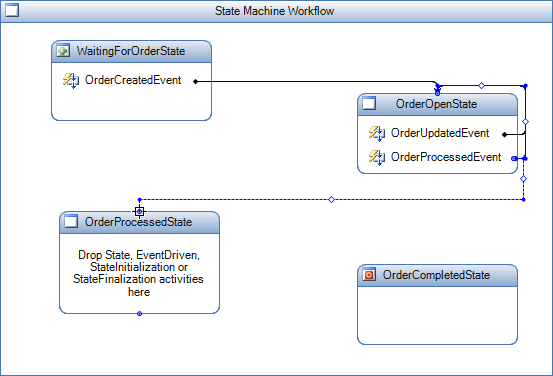


Figure : Current view of workflow

1. After you release the mouse button over the top of the **OrderProcessedState**, click on the blank space in the workflow designer and your workflow should look like this (layout may vary).

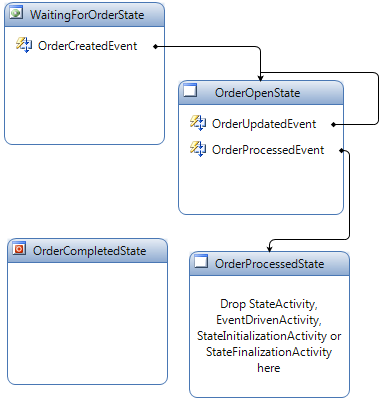


Figure : Current view of workflow

## Task 6 – Define the OrderProcessedState

In task 6 you will configure the **OrderProcessedState** to receive the following events:

1. [**OrderUpdatedEvent**](#_Define_the_OrderUpdated_Event_1) – Transition back to the **OrderOpenState***.*
2. [**OrderCanceledEvent**](#_Define_the_OrderCanceled_Event)–The workflow will terminate as a result of this event.
3. [**OrderShippedEvent**](#_Define_the_OrderShipped_Event)– Transition to the **OrderCompletedState***.*

#### Defining the OrderUpdated Event

1. In the design view for the State Machine Workflow, right-click the **OrderUpdatedEvent** from task 5 in the **OrderOpenState** and select **Copy**.
2. Select the **OrderProcessedState**, right-click and select **Paste**.

This automatically takes you to the design view for the new **EventDriven** activity.

1. Change the **(Name)** property for **eventDrivenActivity1** to **OrderUpdatedEvent2** (as we have already defined an **OrderUpdatedEvent** in the previous task).
2. That is it, as all the activities are configured correctly from task 5.
3. Navigate back to the **State view** in the **State Machine Workflow** by clicking on the **Workflow1** link in the Visual Studio workflow designer.

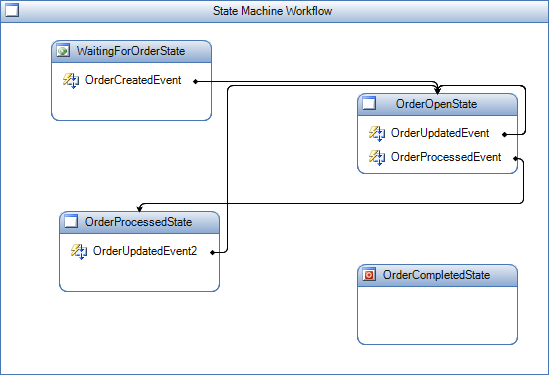


Figure : Current view of workflow

#### Defining the OrderCanceled Event

1. In the design view for the State Machine Workflow, right-click the **OrderCreatedEvent** in the **WaitingForOrderState** and select **Copy**.
2. Select the **OrderProcessedState**, right-click and select **Paste**. This will automatically take you to the design view for the new **EventDriven** activity you pasted.
3. Change the **(Name)** property for **eventDrivenActivity1** to **OrderCanceledEvent**.
4. Select the **HandleExternalEvent** activity and change the **EventName** property to **OrderCanceled**.
5. Set the property **e** to **Workflow1.OrderEvtArgs**
6. Set the **sender** property to **Workflow1.OrderSender** using the activity binding interface.

The properties window should now look like Figure 31.

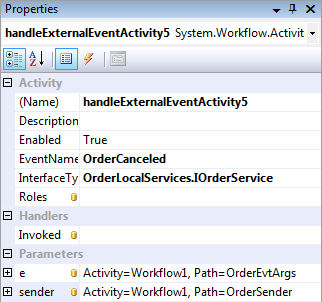


Figure : handleExternalEventActivity5 in the Properties Dialog

1. Select the **SetState** activity, right-click on it and select **Delete**.
2. Add a **Terminate** activity to the workflow definition by dragging and dropping it out of the toolbox into the **OrderCanceledEvent** activity, below the **HandleExternalEvent** activity.

When we receive the cancel event we need to terminate the workflow. The terminate activity will help us do that.

1. With the new **Terminate** activity selected, in the properties window change the **Error** property to **Workflow1.OrderCanceledError** using the activity binding interface.

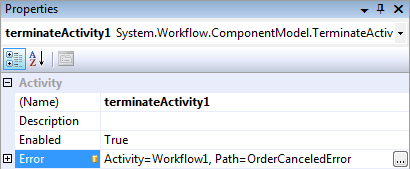


Figure : terminateActivity1 in the Properties Dialog

The workflow should now look like the following:

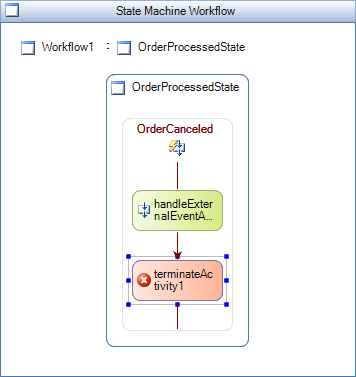


Figure : Current view of the workflow

1. Navigate back to the **State view** in the state machine workflow by clicking on the **Workflow1** link in the Visual Studio workflow designer.

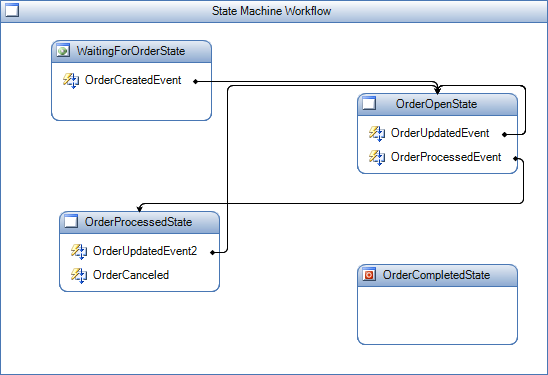


Figure : Current view of workflow

#### Defining the OrderShipped Event

1. In the design view for the State Machine Workflow, right-click the **OrderCreatedEvent** in the **WaitingForOrderState** and select **Copy**.
2. Select the **OrderProcessedState**, right-click and select **Paste**. This will automatically take you to the design view for the new **EventDriven** activity you pasted.
3. Change the **(Name)** property for **eventDrivenActivity1** to **OrderShippedEvent**.
4. Select the **HandleExternalEvent** activity and change the **EventName** property to **OrderShipped**.
5. Set the property **e** to **Workflow1.OrderEvtArgs**
6. Set the **sender** property to **Workflow1.OrderSender** using the activity binding interface.

The properties window should now look like Figure 35.

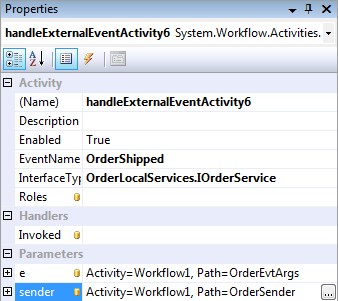


Figure : handleExternalEventActivity6 in the Properties Dialog

1. Make an **Invoked** handler for **handleExternalEventActvity6** by entering **OrderShipped\_Invoked** in the **Invoked** property.

When you press **Enter**, you will be taken to the handler in the code view. Put a break point on the handler as seen in Figure 36.

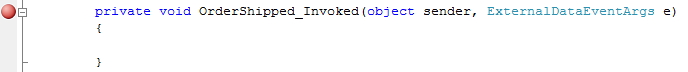


Figure : Breakpoint set in code

1. Switch back to the designer.
2. Select the **SetState** activity set and change the **TargetStateName** property to **OrderCompletedState**.
3. Navigate back to the **State view** in the State Machine Workflow by clicking on the **Workflow1** link in the Visual Studio workflow designer.

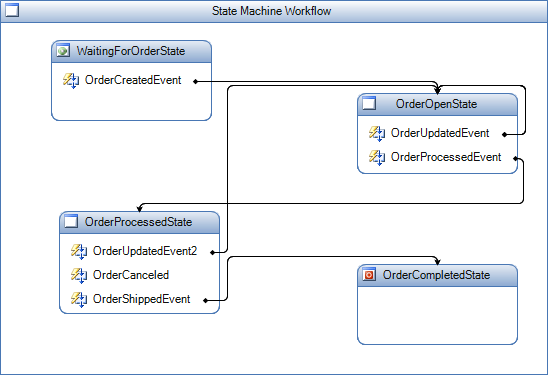


Figure : Current view of workflow

1. The workflow needs another **Invoked** handler for to trap the event when the workflow runs.

Double click on the **OrderCreatedEvent** in the **WaitingForOrderState** state.

1. Select the **HandleExternalEventActivity** and view its properties.
2. Enter **OrderCreated\_Invoked** in the Invoked property.

When you press **Enter**, you will be taken to the handler in the code view.

1. Put a break point on the handler.

## Task 7 – Add the OrderApplication Project

In the previous tasks, you designed the workflow for order management. You now need to create a host application that will call the workflow. The host application will also send the appropriate events to the workflow instance.

For the host application you will create a **Windows Forms** application that provides options for sending events to the workflows. The form also displays the order and the latest status of the workflow associated with the order.

The form will look like .

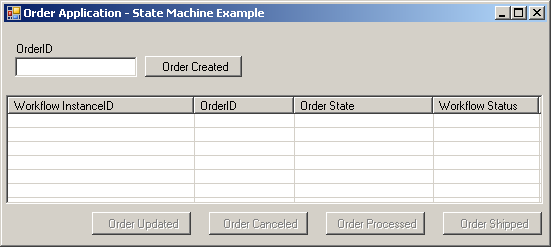


Figure : Windows Form Order Application

Notice the buttons on the form correspond to the events handled by the workflow.

This Windows Forms project has already been created; all you need to do is to include the project in your application by following the steps below:

1. Using **Windows Explorer** navigate to the following directory:

*C:\WF\WF 3.5 Labs\Lab04\Resources*

1. Copy the **OrderApplication-CS** directory to the following location:

*C:\WF\WF 3.5 Labs\Lab04\CSharp\before*

1. Rename:

*C:\WF\WF 3.5 Labs\Lab04\CSharp\before/****OrderApplication-CS***

To:

*C:\WF\WF 3.5 Labs\Lab04\CSharp\before/****OrderApplication***

1. In Visual Studio 2008, in the **OrderWorkflows** solution, add the **OrderApplication** project into the solution by selecting the **File | Add | Existing Project** menu item.
2. Enter or browse to the following **OrderApplication.csproj** file located in the following directory:

*C:\WF\WF 3.5 Labs\Lab04\CSharp\before\OrderApplication*

1. Using the **Solution Explorer tool** window, select the **OrderApplication project**.
2. Expand the **References** folder and remove the **OrderLocalServices** and **OrderWorkflows** references.
3. Add reference to the **OrderLocalServices.dll** in *C:\WF\WF 3.5 Labs\Lab04\Resources\bin*.

(If you need help adding this reference, refer to Task 1.)

1. Add a reference to the **OrderWorkflows** project by right-clicking on the **OrderApplication** project in the solution explorer selecting **Add Reference…**; selecting the **Projects** tab in the dialog and clicking **OK** (as there is only one project to select and it will already be selected).

## Task 8 – Test the State Machine Workflow

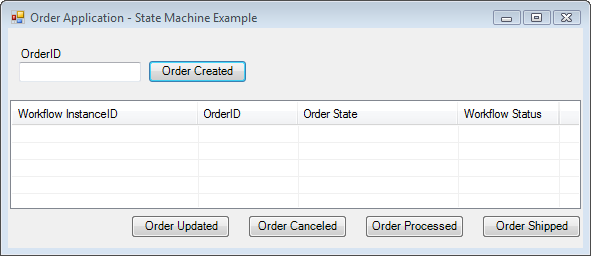
1. Right-click on the **OrderApplication** project in the solution explorer and select **Set as Startup Project**.
2. Build and run the **OrderWorkflows** solution under the Microsoft Visual Studio debugger by pressing **F5** or selecting the **Debug | Start Debugging** menu command.
3. The *OrderApplication* then starts and displays a simple Windows Form for raising the Order Events.   
     
   

Figure : Windows Form Order Application

1. Using the **OrderApplication**, enter a value for the **OrderId** field.
2. Click the **Order Created** button to create a new instance of the **Order State Machine Workflow** and raise an **OrderCreated** event into the new workflow instance.
3. Visual Studio 2008 will then break into debug mode for the **OrderCreated\_Invoked** method. Continue execution of the workflow by selecting the **Debug | Continue** menu item.
4. Information about the new workflow instance will be added to the **ListView** on the form.

|  |
| --- |
| A Note on Moving Forward The Order State column will always be empty during this exercise. We will add functionality to this form in Exercise 2 to provide the Order State. |

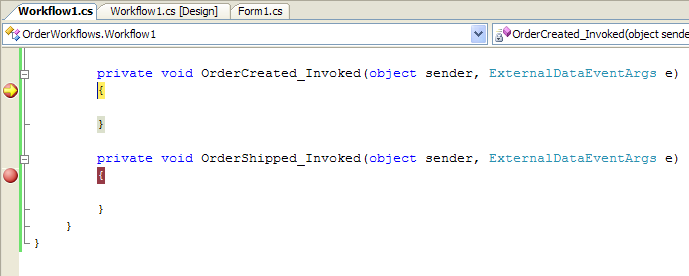


Figure : Breakpoint activated in the code editor

1. Using the **OrderApplication**, select the new item in the **ListView** and click on the **Order Processed** button.
2. Click on the **Order Shipped** button on the **OrderApplication**.

|  |
| --- |
| **NOTE:** If you click the buttons on the Order Application out of order and raise an event that is not expected by the State Machine Workflow, then you will either see an exception or the workflow will do nothing. We will learn how to inspect a State Machine Workflow’s current state in Exercise 2. |

1. Visual Studio 2008 will then go into debug mode for the **OrderShipped\_Invoked** method.
2. Continue execution of the workflow by selecting the **Debug | Continue** menu item.
3. Finally, close the **OrderApplication** to stop debugging.

# Exercise 2 – Use the StateMachineTracking Service

In Exercise 1 you learned how to author a state machine workflow. You tested the workflow with a simple Windows Forms application, called the **OrderApplication**, which allowed you to raise events for an order so they could be handled by the workflow. However, you might have noticed that the **OrderApplication** allowed you to raise any order event regardless of the state of the order workflow and the events the workflow could receive.

In this exercise you’re going to modify a simple version of the **OrderApplication** to use a sample runtime service called the **StateMachineTracking** service. The **StateMachineTracking** service allows a host to inspect the state of a state machine workflow and receive key events during its lifetime.

For example, using the **StateMachineTracking** service a host application can receive a **StateChanged** event whenever the workflow performs a state transition using the **SetState** activity. You will use the **StateMachineTracking** service to enhance the **OrderApplication** to display the current state for each order record in the **ListView**.

You will also customize the **OrderApplication** so that a user can only raise order events that are valid based on the order’s current state. For example, if an order is in a processed state, then the **OrderApplication** will only allow the user to raise an **OrderShipped** or an **OrderUpdated** event.

## Task 1 – Add a reference to the StateMachineTracking Service

Before we can begin using the **StateMachineTracking** service, we also need to add a reference to the *StateMachineTracking.dll*.

1. Right click on the **References** folder for the **OrderApplication** project in Solution Explorer and
2. Select **Add Reference** from the context menu.
3. In the **Add Reference dialog** window, select the **Browse** tab.
4. Enter or browse to the *StateMachineTracking.dll* assembly located in the following directory:

*C:\WF\WF 3.5 Labs\Lab04\Resources\bin*

1. Press the **OK** button to close the **Add Reference dialog** window and add the new reference.

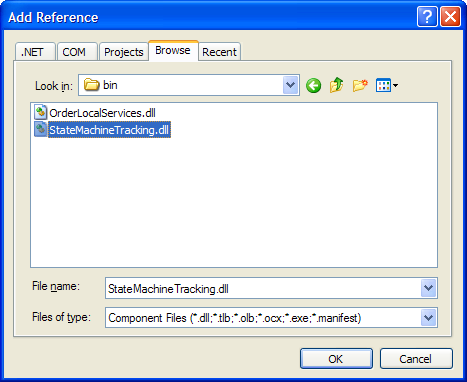


Figure : Add Reference dialog

## Task 2 – Modify the code in the OrderApplication

1. Right click on **Form1.cs** in Solution Explorer and select **View Code** from the context menu.
2. Add a using statement for the **StateMachineTracking** namespace at the top of the code file as shown in the highlighted code below:

|  |
| --- |
| **using System.Workflow.Activities;**  **using System.Threading;**  **using StateMachineTracking;**  **namespace OrderApplication**  **{**  **. . .** |

1. Add a member-level variable in the **Form1** class for holding a reference to the **StateMachineTrackingService** as shown in the code below:

|  |
| --- |
| **namespace OrderApplication**  **{**  **public partial class Form1 : Form**  **{**  **private StateMachineTrackingService stateMachineTrackingService;**  **private WorkflowRuntime \_WFRuntime;**  **private OrderLocalServices.OrderService \_OrderService;**  **. . .** |

1. Add another member-level variable in the **Form1** class for managing a generic dictionary of **StateMachineInstance** objects:

|  |
| --- |
| **namespace OrderApplication**  **{**  **public partial class Form1 : Form**  **{**  **private StateMachineTrackingService stateMachineTrackingService;**  **private Dictionary<string, StateMachineInstance>**  **stateMachineInstances;**  **private WorkflowRuntime \_WFRuntime;**  **private OrderLocalServices.OrderService \_OrderService;**  **. . .** |

1. Modify the code in the **Form\_Load** method to create a new instance of a generic dictionary for the type **StateMachineInstance** class and assign the new list to the member-level variable.

You also need to disable all of the buttons that can be used to raise events. Add this code **before** the call to **StartWorkflowRuntime()** as shown in the code below.

|  |
| --- |
| **private void Form1\_Load(object sender, EventArgs e)**  **{**  **this.stateMachineInstances = new Dictionary<string, StateMachineInstance>();**  **this.DisableButtons();**  **// Create and start the WorkflowRuntime**  **this.StartWorkflowRuntime();**  **}** |

1. Modify the code in the **StartWorkflowRuntime** method to create a new instance of the **StateMachineTrackingService** class and add the new instance to the workflow runtime as shown in the highlighted code below.

|  |
| --- |
| **private void StartWorkflowRuntime()**  **{**  **// Create a new Workflow Runtime for this application**  **\_WFRuntime = new WorkflowRuntime();**  **// Create EventHandlers for the WorkflowRuntime**  **\_WFRuntime.WorkflowTerminated +=**  **new EventHandler<WorkflowTerminatedEventArgs>**  **(WorkflowRuntime\_WorkflowTerminated);**  **\_WFRuntime.WorkflowCompleted +=**  **new EventHandler<WorkflowCompletedEventArgs>**  **(WorkflowRuntime\_WorkflowCompleted);**  **stateMachineTrackingService = new StateMachineTrackingService(\_WFRuntime);**  **// Add a new instance of the OrderService to the runtime**  **ExternalDataExchangeService dataService =**  **new ExternalDataExchangeService();**  **\_WFRuntime.AddService(dataService);**  **\_OrderService = new OrderLocalServices.OrderService();**  **dataService.AddService(\_OrderService);**  **}** |

1. Modify the code in the **StartOrderWorkflow** method to use the **StateMachineTracking** Service. **Delete** the code that starts a new workflow. Instead, call the **StartWorkflow** method on the **StateMachineTracking** Service so it can intercept the events that occur from the workflow. We also need to hook onto the **StateChanged** event from the **StateMachineInstance**. Finally, we will add the **StateMachineInstance** object to our member-level list.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_StartOrderWorkflow |
| **private System.Guid StartOrderWorkflow(string strOrderID)**  **{**  **// Create a new GUID for the WorkflowInstanceId**  **System.Guid WorkflowInstanceId = System.Guid.NewGuid();**  **StateMachineInstance stateMachineInstance =**  **stateMachineTrackingService.RegisterInstance(**  **typeof(OrderWorkflows.Workflow1),**  **WorkflowInstanceId);**  **stateMachineInstance.StateChanged +=**  **new EventHandler<ActivityEventArgs>**  **(StateMachineInstance\_StateChanged);**  **stateMachineInstance.StartWorkflow();**  **stateMachineInstances.Add(WorkflowInstanceId.ToString(),**  **stateMachineInstance);**  **// Return the WorkflowInstanceId**  **return WorkflowInstanceId;**  **}** |

1. Create a helper method named **GetSelectedStateMachineInstance**.

This method will get the **WorkflowInstanceId** for the selected record in the **ListView** and lookup the corresponding **StateMachineInstance** object from our member-level list.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_GetSelectedStateMachine |
| **private StateMachineInstance GetSelectedStateMachineInstance()**  **{**  **// Get the WorkflowInstanceId for the selected item**  **string workflowInstanceId = this.lstvwOrders.SelectedItems[0].Text;**  **// Return the StateMachineInstance object**  **return stateMachineInstances[workflowInstanceId];**  **}** |

1. Use the following snippet to create the **EnableButtons** method to enable the buttons on this form based on the messages or events allowed into the state machine workflow instance.

First, check and make sure that a record is selected in the **ListView**; then check to make sure that the workflow is still running.

Finally determine whether the workflow accepts a specific message (such as **OrderShipped**). This functionality is provided through the **StateMachineInstance** class included with the **StateMachineTrackingService**.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_EnableButtons |
| **private void EnableButtons()**  **{**  **if (this.lstvwOrders.SelectedItems.Count == 0)**  **{**  **return;**  **}**  **string workflowStatus = this.lstvwOrders**  **.SelectedItems[0].SubItems[3].Text;**  **if ((workflowStatus.Equals("Completed")) ||**  **(workflowStatus.Equals("Terminated")))**  **{**  **return;**  **}**  **// Return the StateMachineInstance object**  **StateMachineInstance stateMachineInstance =**  **this.GetSelectedStateMachineInstance();**  **if (stateMachineInstance.CurrentState == null)**  **{**  **return;**  **}**  **if (stateMachineInstance.MessagesAllowed.Contains("OrderCanceled"))**  **this.btnOrderCanceled.Enabled = true;**  **if (stateMachineInstance.MessagesAllowed.Contains("OrderProcessed"))**  **this.btnOrderProcessed.Enabled = true;**  **if (stateMachineInstance.MessagesAllowed.Contains("OrderShipped"))**  **this.btnOrderShipped.Enabled = true;**  **if (stateMachineInstance.MessagesAllowed.Contains("OrderUpdated"))**  **this.btnOrderUpdated.Enabled = true;**  **}** |

1. Create a helper method called **DisableButtons**.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_DisableButtons |
| **private void DisableButtons()**  **{**  **this.btnOrderCanceled.Enabled = false;**  **this.btnOrderProcessed.Enabled = false;**  **this.btnOrderShipped.Enabled = false;**  **this.btnOrderUpdated.Enabled = false;**  **}** |

1. Modify the code in the **btnOrderEvent\_Click** method to disable all of the buttons on the form before raising an event. Add the line of code highlighted in yellow.

|  |
| --- |
| **private void btnOrderEvent\_Click(object sender, EventArgs e)**  **{**  **// Get the Name for the button that was clicked**  **string strButtonName = ((Button)sender).Name;**  **// Get the WorkflowInstanceId for the selected order**  **System.Guid WorkflowInstanceId = this.GetSelectedWorkflowInstanceID();**  **// Get the OrderID for the selected order**  **string strOrderId = this.GetSelectedOrderId();**  **this.DisableButtons();**  **switch (strButtonName)**  **{**  **. . .**  **}**  **}** |

1. Create an **UpdateButtonStatus** method to disable and then enable the buttons on this form. This method needs to use a delegate to invoke itself if the code is executing on a different thread than the thread used to create the button controls originally.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_UpdateButtonStatus |
| **private void UpdateButtonStatus()**  **{**  **if (this.InvokeRequired)**  **{**  **UpdateButtonStatusDelegate updateButtonStatus =**  **new UpdateButtonStatusDelegate(this.UpdateButtonStatus);**  **this.Invoke(updateButtonStatus);**  **}**  **else**  **{**  **DisableButtons();**  **EnableButtons();**  **}**  **}** |

1. Create the **StateMachineInstance\_StateChanged** method to handle the **StateChanged** event that occurs when the state in our **Order State Machine Workflow** is changed.

The method updates the record in the **ListView** and the status of the buttons on the form based on the current state unless it is the completed state.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_StateChanged |
| **protected void StateMachineInstance\_StateChanged(object sender,**  **ActivityEventArgs e)**  **{**  **StateMachineInstance stateMachineInstance = (StateMachineInstance)sender;**  **if (e.QualifiedId != stateMachineInstance.CompletedState)**  **{**  **WorkflowInstance workflowInstance =**  **stateMachineInstance.WorkflowInstance;**  **this.UpdateListItem(workflowInstance, e.QualifiedId.ToString(),**  **"Running");**  **this.UpdateButtonStatus();**  **}**  **}** |

1. Modify the code in the **lstvwOrders\_ItemSelectionChanged** method to update the status for the buttons on this form when a record is selected in the **ListView**. This method disables all buttons if the item is deselected.

|  |
| --- |
| **Snippet:** WFLab04\_Ex02\_Task2\_ItemSelectionChanged |
| **private void lstvwOrders\_ItemSelectionChanged(object sender,**  **ListViewItemSelectionChangedEventArgs e)**  **{**  **if (e.Item.Selected)**  **{**  **this.UpdateButtonStatus();**  **}**  **else**  **{**  **this.DisableButtons();**  **}**  **}** |

## 

## Task 3 – Test the OrderApplication and State Machine Tracking

1. Remove the breakpoints inserted in the previous exercise by selecting from the menu:  
   **Debug | Delete All Breakpoints**
2. Compile and run the solution under the Visual Studio debugger by pressing **F5** or selecting the **Debug | Start Debugging** menu command.
3. You will then see the **OrderApplication**.
4. Enter a value for the **OrderId** field and press the **Order Created** button to create a new instance of the workflow and raise an **OrderCreated** event.

You can repeat this process to create several instances of the Order State Machine Workflow.

1. As you create the Orders, you should see the **Order State** value for each record quickly change from **WaitingForOrderState** to **OrderOpenState**.
2. When you select a record, you should notice that the **Order Updated** and **Order Processed** buttons are enabled.
3. Press the **Order Processed** button to raise an **OrderProcessed** event to the workflow.
4. After a few seconds, you should notice that the **Order State** has changed to **OrderProcessState** and now when the record is selected only the **Order Updated**, **Order Shipped**and **Order Canceled** buttons are enabled.

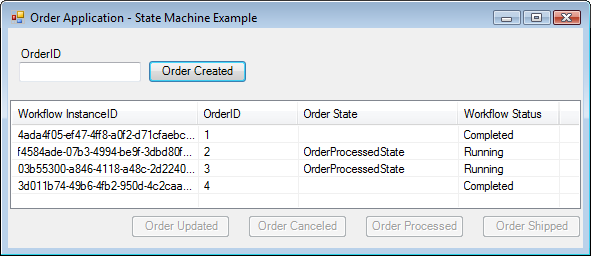


Figure : Several orders in various states

## 

# Lab Summary

This lab introduced the model of **State Machine Workflows**, available when working with **Windows Workflow Foundation**. State Machine Workflows differ from Sequential Workflows in that the events within the workflow do not occur in a specific order, but occur in response to events within the workflow.

In this lab you performed the following exercises:

Create the Order State Machine Workflow.

Use the *StateMachineTracking* Service.

The exercises demonstrated how to model a **State Machine Workflow** in **Windows Workflow Foundation** using the Visual Studio workflow designer. The application you created utilized the **State**, **SetState**, and **EventHandler** activities. You also learned how to interact with a **State Machine Workflow** from the host application using the sample **StateMachineTracking** service.