

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 01 – Getting Started with Windows Workflow Foundation

|  |  |
| --- | --- |
| Lab version: | 2.0.0 |
| Last updated: | 12/3/2008 |

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.

© 2005 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 3](#_Toc218632795)

[Objectives 3](#_Toc218632796)

[System Requirements 4](#_Toc218632797)

[Setup 4](#_Toc218632798)

[Physical Folder Structure 4](#_Toc218632799)

[Code Snippets 4](#_Toc218632800)

[Starting Material 5](#_Toc218632801)

[Acronyms Used in this Lab 5](#_Toc218632802)

[Scenario 5](#_Toc218632803)

[Exercise 1 – Create a Hello World Workflow 5](#_Toc218632804)

[Task 1 – Create a new Workflow Project 5](#_Toc218632805)

[Task 2 – Add a code activity 8](#_Toc218632806)

[Task 3 – Run the new workflow with Debugging 10](#_Toc218632807)

[Exercise 2 – Receiving Data into the Workflow using Parameters 13](#_Toc218632808)

[Task 1 – Define parameters for the workflow 13](#_Toc218632809)

[Task 2 – Modify the Code Activity 15](#_Toc218632810)

[Task 3 – Execute the modified workflow 17](#_Toc218632811)

[Exercise 3 – Using the IfElse activity, declarative conditions and custom activities 23](#_Toc218632812)

[Task 1 – Create a new Workflow Project 23](#_Toc218632813)

[Task 2 – Add the Expense Reporting Interface to the Project 27](#_Toc218632814)

[Task 3 – Add the ExpenseReportSubmitted activity 28](#_Toc218632815)

[Task 4 – Add the pre-built AutoApprove activity to the workflow 33](#_Toc218632816)

[Task 4 – Add a Rule Condition to the workflow 35](#_Toc218632817)

[Task 5 – Add the logic to approve or reject the expense report 38](#_Toc218632818)

[Task 6 – Configure the ExpenseWorkflows project for debugging 41](#_Toc218632819)

[Task 7 – Test the Expense Reporting Workflow 43](#_Toc218632820)

[Exercise 4 – Using Listen, Delay, and event-based custom activities 49](#_Toc218632821)

[Task 1 – Add activities for requesting manager approval 49](#_Toc218632822)

[Task 2 – Add a Listen activity for waiting for the Expense Report Review 53](#_Toc218632823)

[Task 3 – Approve or Reject the Expense Report 56](#_Toc218632824)

[Task 4 – Test the Expense Reporting Workflow 64](#_Toc218632825)

[Lab Summary 69](#_Toc218632826)

# Overview

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

This lab introduces .NET developers to Windows Workflow Foundation (WF), and the key concepts behind the technology. The lab assumes no prior experience with WF, and uses Visual Studio 2008 to introduce the basic concepts of WF: constructing sequential workflows, using the WF graphical designer to manage basic flow control, create branching conditions, and interact with external applications using events. As we explore WF, the lab walks you through how to use the debugger to set and use breakpoints in code as well as the WF graphical designer.

## Objectives

The objective of this lab is to introduce the key Windows Workflow Foundation workflow concepts.

After completing this lab, you will be able to:

* Author Sequential Workflows using the Visual Studio 2008 designer for Windows Workflow Foundation
* Configure and use the Visual Studio 2008 debugger to debug your workflows
* Receive parameters into a workflow
* Receive an event to activate a new instance of a workflow
* Define conditions
* Configure basic activities including **IfElse**, **Delay**, **Listen**, and **Code**
* Utilize custom activities for receiving events into a workflow

|  |
| --- |
| What is a Workflow? A workflow is a model of a human or system process which is defined as a map of activities. An activity is a step in a workflow and is the unit of execution, re-use and composition for a workflow. A workflow is designed by laying out activities and is compiled to a .NET assembly that is executed on the workflow runtime and the Common Language Runtime (CLR). |

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:

* **VB** – 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

* Exercises 1 and 2 presents a simple “HelloWorld” application to illustrate fundamentals.
* Exercises 3 and 4 model a scenario in which a workflow is used to manage the submission and processing of expense reports within an organization.

# Exercise 1 – Create a Hello World Workflow

In this exercise, you will create a very simple “Hello World” workflow using the Visual Studio 2008 designer for Windows Workflow Foundation, referred to as the Visual Studio workflow designer in the remainder of this document. This is a visual designer in Visual Studio for creating workflows.

The Hello World workflow will be a sequential workflow that includes a single code activity. The code activity will be used to write the phrase “Hello, World!” to the console at runtime. You will explore the workflow definition and the code-beside for the workflow.

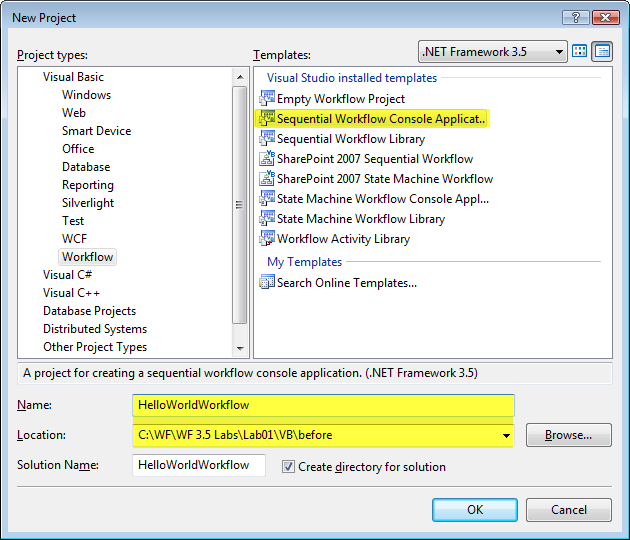
Finally, you will execute your workflow using a simple host and debug your workflow using Visual Studio 2008.

## Task 1 – Create a new Workflow Project

1. Open Visual Studio 2008 by going to:   
   **Start Menu | Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**
2. In Visual Studio 2008, select the menu command:   
   **File | New | Project**
3. Visual Studio 2008 will display the **New Project** window.
4. In the **New Project** dialog, expand **Visual Basic | Workflow** in the **Project Types** tree on the left side.
5. Select the template named **Sequential Workflow Console Application** and enter the following values:

**Name:** *HelloWorldWorkflow*

**Location:** *C:\WF\WF 3.5 Labs\Lab01\VB\before*



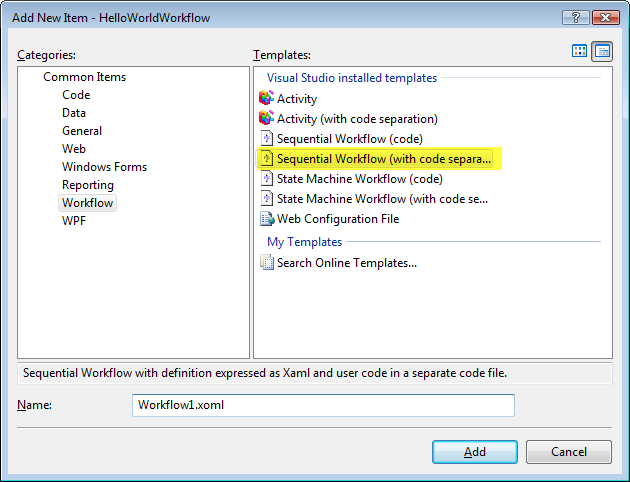
**Figure 7: Add New Sequential Workflow Application Dialog**

1. Click **OK**

You should now have a new solution and workflow project

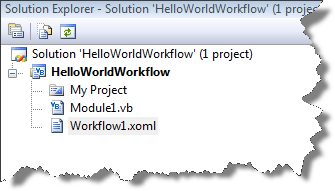
#### Replace the files Visual Studio created by default.

1. By default, Visual Studio creates a *Workflow1* code file. Remove this file.
   1. Click on *Workflow1* code file and press the **DEL** key.
   2. Click the **OK** button to confirm.
2. **Right-click** on the **HelloWorldWorkflow** project in **Solution Explorer** and select **Add | New Item** from the context menu.
3. In the **Add New Item** dialog, select the item template **Sequential Workflow (with code separation)** and click the **Add** button.



**Figure 8: Add New Sequential Workflow Item Dialog**

*Workflow1.xoml* is added to the project.



**Figure 9: Solution Explorer**

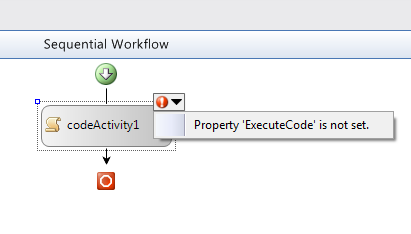
## Task 2 – Add a code activity

A basic sequential workflow now exists in the project. It needs some WF Activities in order to actually be useful. Activities contain the actual work that gets done in workflows.

1. **Double-click** on the *Workflow1.xoml* file in **Solution Explorer** to open the Visual Studio workflow designer.
2. Select the **View | Toolbox** menu command to display the Toolbox. It shows the activities available for use in the workflow.

**Note:** You may need to expand the Windows Workflow categories in the Toolbox to view the Windows Workflow Foundation activities.

1. Select the **Code** activity from the Toolbox.
2. Drag and drop the **Code** activity to the design surface for the Sequential Workflow.
3. Notice the red exclamation mark on the new **Code** activity. **Click** the exclamation mark to view the smart tag. It shows the reason it is being displayed.

****

**Figure 10: Code Activity Warning**

1. Selecting the smart tag will cause the **Properties** tool window to be displayed with the **ExecuteCode** property highlighted.
2. Enter the value codeActivity1\_CodeHandler for the **ExecuteCode** property and press **Enter**. This will cause a new method or ***event handler*** to be added to the code-beside class for the workflow.
3. Visual Studio will automatically open the *Workflow1.xoml.vb* code file and display the new codeActivity1\_CodeHandler method.   
   **Note:** You can switch from the design view to the code-beside by double-clicking on the activity or by selecting the **View | Code** menu item.
4. The code-beside class is a class that inherits from the **SequentialWorkflowActivity** base class. This code-beside class will have the same name as the workflow. You should have the following code in your code-beside class:

|  |
| --- |
| **Public Class Workflow1**  **Inherits SequentialWorkflowActivity**  **Private Sub codeActivity1\_CodeHandler(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs)**  **End Sub**  **End Class** |

1. Insert the following code in the codeActivity1\_CodeHandler method to output the string “Hello, World!” to the console:

|  |
| --- |
| **Note:** Added code is shown in yellow highlight. |
| **Public Class Workflow1**  **Inherits SequentialWorkflowActivity**  **Private Sub codeActivity1\_CodeHandler(ByVal sender As System.Object, ...**  Console.WriteLine("Hello, World!")  **End Sub**  **End Class** |

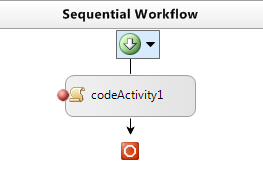
1. Build your project (**Build | Build Solution** menu command or **CTRL+Shift+B**). Verify that there are no compiler errors or warnings.

## Task 3 – Run the new workflow with Debugging

This exercise uses the simple host in Module.vb that comes with the project template that was selected. Alternatively, you could build your own workflow host application by using the workflow runtime APIs defined in the System.Workflow.Runtime namespace.

#### Add a breakpoint to the workflow

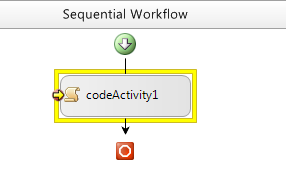
1. Open the Visual Studio workflow designer (**Double-click** the XOML code file) in the **Solution Explorer**.
2. Click on the **codeActivity1** activity.
3. Right-click on **codeActivity1** and choose **Breakpoint** | **Insert Breakpoint**.
4. You should now see a red circle to the left of **codeActivity1**. This is the indicator that a breakpoint has been set on the activity.

****

**Figure 11: Breakpoint set on an activity**

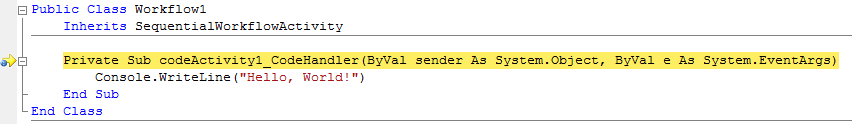
#### Running the workflow with the debugger

1. Compile and run the solution under the Visual Studio debugger (**F5** or the **Debug | Start Debugging** menu command).
2. The code in Module.vb will start an instance of Workflow1. That instance will break in the debugger when it gets to the code activity. The yellow box indicates the activity where the debugger is stopped.

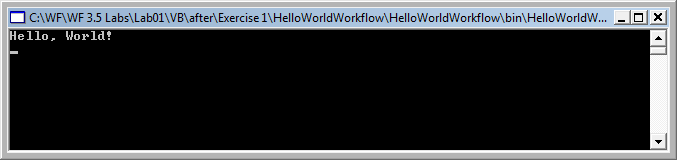


**Figure 12: Workflow stopped on debugger breakpoint**

1. Step into the code1\_CodeHandler method (**Debug** | **Step** **Into** from the menu or **F11**).

****

**Figure 13: The debugger stopped on a breakpoint**

1. Press **F11** a couple of more times and you should see “Hello, World!” written out into the test host’s console window.  
     
   ****

**Figure 14: Hello World**

1. Press **F5** to continue program execution and the workflow will now complete and the program finishes.

# Exercise 2 – Receiving Data into the Workflow using Parameters

In this exercise, you will modify the workflow we created in Exercise 1 to receive data into our workflow when it is started. You will also modify the code in our Code activity’s handler to display the input values.

Finally, you will change our startup project to use a Windows Forms (WinForms) host that will allow us to enter the input values and create an instance of our workflow.

|  |
| --- |
| Providing Data to Workflows There are two general approaches for providing data to a workflow when it is started: Parameters and events.  With parameters, a list of parameter names and types are defined with the workflow. These parameter values are passed in by a host when it starts a new instance of the workflow type.  With events, workflow authors add an activity that receives an event and data associated with the event. Events are generally specific to the host and custom activities that have been designed to handle the event.  We’ll use parameters in this exercise and learn how to use event-based activities in Exercise 3. |

## Task 1 – Define parameters for the workflow

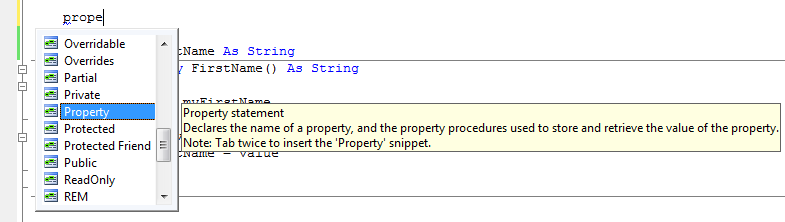
Workflow properties may be used to pass data into the workflow. The properties are no different than standard .NET properties on any class file.

#### Define properties on the workflow

1. Open *Workflow1.xoml* in the Visual Studio workflow designer (double-click the file in **Solution Explorer**).
2. Open the code of the workflow class.

Right-click the designer background in the Visual Studio workflow designer and select **View Code**, or use the **View | Code** menu item.

1. Inside the *Workflow1* class, type “prope” and select **Property** from the **IntelliSense** menu. Press **TAB** twice to insert the snippet.

****

**Figure 15: Using the Property Code Snippet**

1. Fill in each of the highlighted areas in the snippet to create an string property called FirstName.
2. Use the same technique to include another string property called LastName.

|  |
| --- |
| **Private Sub codeActivity1\_CodeHandler(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs)**  **End Sub**  Private myFirstName As String  Public Property FirstName() As String  Get  Return myFirstName  End Get  Set(ByVal value As String)  myFirstName = value  End Set  End Property  Private myLastName As String  Public Property LastName() As String  Get  Return myLastName  End Get  Set(ByVal value As String)  myLastName = value  End Set  End Property |

## Task 2 – Modify the Code Activity

The work done by the workflow will be simple. It will pop up a Message Box. To use the MessageBox class, a reference must be added to the Windows Forms assembly.

#### Add a reference to the System.Windows.Forms

1. **Right-click** the **HelloWorldWorkflow** project in **Solution Explorer.**
2. Select **Add Reference** from the context menu.
3. On the **.NET** tab in the Add Reference dialog, select **System.Windows.Forms** from the list of assemblies and click the **OK** button.

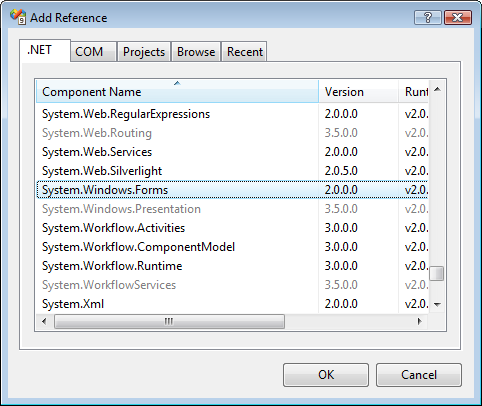


Figure : Add Reference Dialog

#### Pop up the Message Box in the Code activity

1. **Open** *Workflow1.xoml.vb* for editing (**double-click** the **codeActivity1** activity). Navigate to the method codeActivity1\_CodeHandler. This method is the entry point for the code activity in the workflow.
2. Modify the code in the codeActivity1\_CodeHandler method to get the parameter values and display a Message Box with the FirstName and LastName values:

|  |
| --- |
| **Private Sub codeActivity1\_CodeHandler(ByVal sender As System.Object, \_**  **ByVal e As System.EventArgs)**  System.Windows.Forms.MessageBox.Show(myFirstName & " " & myLastName)  **End Sub** |

## Task 3 – Execute the modified workflow

It is time to test the modified workflow. Instead of using a console host application as we did in Exercise 1, you will take advantage of a preexisting Windows Forms application with some pre-written code as a host for the workflow.

#### Add the WinForm project to the solution

1. In Visual Studio 2008, with the **HelloWorldWorkflow** solution open, add an existing project. **WinFormTestHost** to the solution by selecting the **File | Add | Existing Project** menu item.
2. In the Add Existing Project browser dialog, select the *WinFormTestHost.csproj* file from the folder:  
   *C:\WF\WF 3.5 Labs\Lab01\resources\WinFormTestHost*
3. Click **Open**
4. The project is added to the **HelloWorldWorkflow** solution.

#### Preparing the WinForm application

1. Using the **Solution Explorer** tool window, **Right-click** on the **WinFormTestHost** project and select **Set as Startup Project** from the context menu.

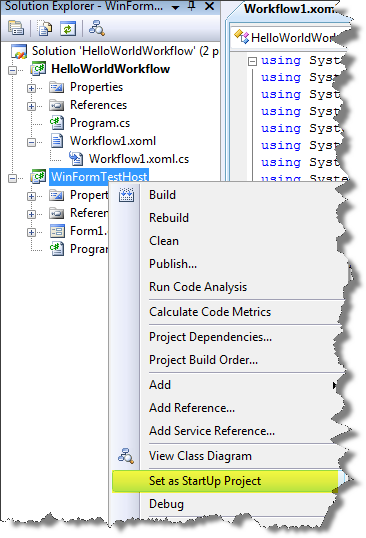


Figure : Set the WinFormTestHost project as StartUp Project

1. Now we need to make sure that the **WinFormTestHost** application can use the new workflow. Add a reference to the **HelloWorldWorkflow** project by **right-click**ing on the **References** folder for the **WinFormTestHost** in **Solution Explorer** and selecting **Add Reference** from the context menu.
2. In the **Add Reference** dialog window, select the **Projects** tab.
3. Select the **HelloWorldWorkflow** project from the list and press the **OK** button to close the **Add Reference** dialog window and add the new reference.

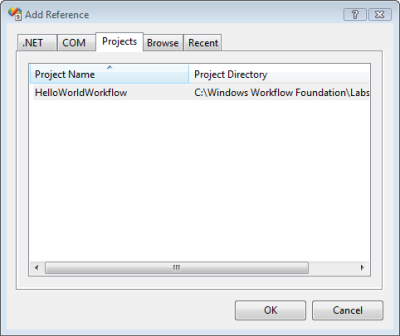


Figure : Adding a Project reference

1. We also need to ensure that the **WinFormTestHost** has references to the WF system assemblies. Add these references by **right-clicking** on the **References** folder for the **WinFormTestHost** in **Solution Explorer** and selecting **Add Reference** from the context menu.
2. In the **Add Reference** dialog window, select the **.NET** tab, scroll down and select the entries shown below and click **OK**.

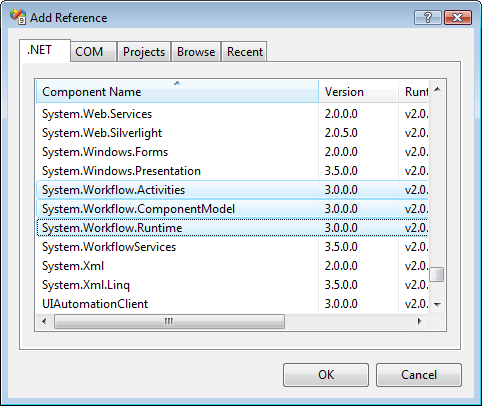


Figure : Adding references to WF assemblies

1. Perform a test compile by pressing **CTRL+Shift+B**. The solution should compile with no errors.

#### Understanding the WinForm project

The **WinFormTestHost** project is a C# Windows Forms project. A service may be hosted in many kinds of applications and this is only a simple example of one scenario.

1. Open the Form1.cs code file (**right-click** *Form1.cs* **| View Code**).
2. Note the member variable wr, which is an instance of the WF WorkflowRuntime class. This class will be used to instantiate and run the workflow.
3. Note the following code, used to gather information from the Windows Form and populate a Dictionary to be used as parameters to the workflow.

|  |
| --- |
| **Dictionary<string, object> parameters = new Dictionary<string, object>();**  **parameters.Add("FirstName", txtFirstName.Text);**  **parameters.Add("LastName", txtLastName.Text);** |

1. Lastly, note the following code that actually instantiates and starts the workflow.

|  |
| --- |
| **WorkflowInstance instance =**  **wr.CreateWorkflow(typeof(HelloWorldWorkflow.Workflow1), parameters);**  **instance.Start();** |

1. This code will be invoked when the button is clicked on the Windows Form.

#### Set a breakpoint in the Code activity handler

1. **Double-click** *Workflow1.xoml* in **Solution Explorer**.
2. **Double-click** **codeActivity1** in the visual designer.
3. Set a breakpoint in the codeActivity1\_CodeHandlermethod.

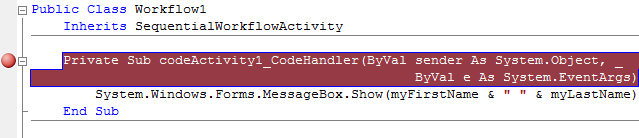


Figure : Break Point set on handler

#### Run the workflow under with the debugger

1. Compile and run the solution under the debugger (Press **F5)**.
2. The **WinFormTestHost** will start and display a window containing fields for entering a First and Last Name.
3. Enter values in the **First name** and **Last name** fields
4. Click the **Start Workflow** button to instantiate a new instance of the HelloWorldWorkflow.Workflow1 and pass in the First name and Last name values as parameters.

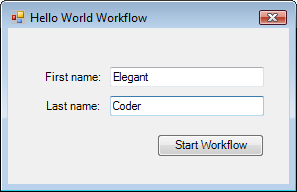


Figure : Hello World Windows Form Dialog

1. Visual Studio will then break into debug mode for the codeActivity1\_CodeHandler method. Continue execution of the workflow by pressing **F5**.

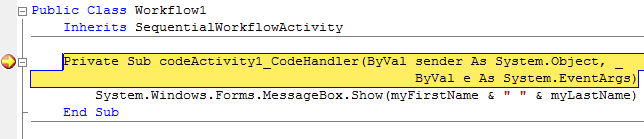


Figure : Workflow stopped at breakpoint

1. When the code activity finishes executing you will see a message box with the first name and last name that were passed as parameters to the workflow.

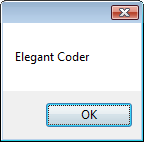


Figure : MessageBox displayed

1. Close the **WinFormTestHost** application to stop the host and stop the debugger.
2. Close Visual Studio

# Exercise 3 – Using the IfElse activity, declarative conditions and custom activities

In this exercise you will learn how to utilize the **IfElse** activity and a declarative condition in the workflow to perform conditional processing. This enables designing conditional logic into a workflow without writing code. You will also learn how to use custom activities within a workflow, which enables reusing custom Activities.

Build a simple sequential workflow for processing an expense report. The workflow will receive an event when the expense report is submitted. The event arguments will contain the actual expense report to process.

If the amount of the expense report is less than $1000, then automatically approve it. However, if it is $1000 or greater, automatically reject the expense report.

|  |
| --- |
| A Note on Code in the Lab To simplify this lab and focus on workflow authoring concepts, you will be provided with the custom Expense Reporting activities, workflow host, and Windows Application for submitting an expense reports to the workflow.  Because some of the parts of this lab exercise are predefined you must be sure to use the project name **ExpenseWorkflows** as listed below. Otherwise your lab will not work as expected |

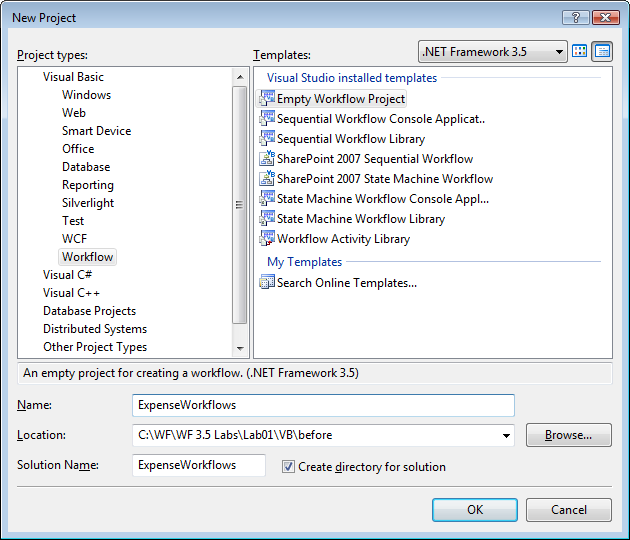
## Task 1 – Create a new Workflow Project

#### Creating a new Sequential Workflow

1. Open Visual Studio 2008 by going to the **Start Menu | Programs | Microsoft Visual Studio 2008 | Microsoft Visual Studio 2008**
2. In Visual Studio 2008, select the **File | New | Project** menu command.
3. Visual Studio will display the **New Project** dialog window.
4. In the **New Project** dialog window, expand **Visual Basic | Workflow** in the **Project Types** tree on the left side.
5. Select the template named **Empty Workflow Project** and enter the following values for the name and location:

**Name:** *ExpenseWorkflows*

**Location:** *C:\WF\WF 3.5 Labs\Lab01\VB\before*

****

**Figure 24: Adding new Empty Workflow Project**

1. You should now have a new solution and workflow project. Now we need to add a new **Sequential Workflow** to the project.

#### Adding a workflow to the project

1. Select the **Project | Add Sequential Workflow** menu item. This will cause Visual Studio to display the **Add New Item** dialog window.

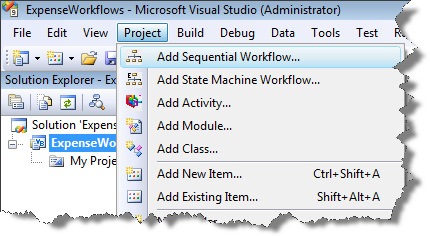


Figure : Adding a new Sequential Workflow

1. In the **Add New Item** dialog window, select the workflow template **Sequential Workflow (with code separation)** and use the default workflow name of Workflow1.xoml*.*

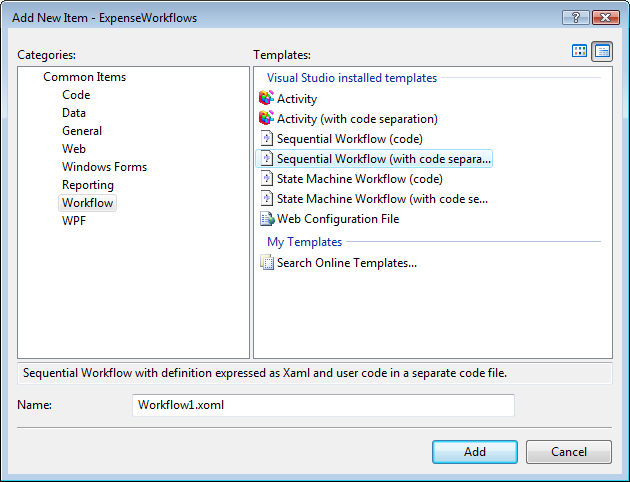


Figure : Add New Item Dialog

1. *Workflow1.xoml* and *Workflow1.xoml.vb* will now be added to your **ExpenseWorkflows** project. *Workflow1.xoml* contains the XML workflow markup that represents the workflow definition. *Workflow1.xoml.vb* contains the code-beside class for the workflow. You will see this file in the folder on your hard drive, but Visual Studio does not show it to you unless you want to see it via **View | Code**.

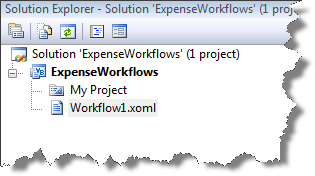


Figure : Solution Explorer after Sequential Workflow

## Task 2 – Add the Expense Reporting Interface to the Project

We will use the **HandleExternalEvent** and **CallExternalMethod** activities to talk to a .NET assembly that already contains the logic needed to process expense reports. This assembly is provided in a pre-existing DLL.

#### Adding a reference to the provided assembly

1. With the **ExpenseWorkflows** project open, select the **Project | Add Reference** menu item.
2. In the **Add Reference** dialog window, select the **Browse** tab.
3. Enter or browse to the *ExpenseLocalServices.dll* assembly located in the following directory:

*C:\WF\WF 3.5 Labs\Lab01\resources\Lib*

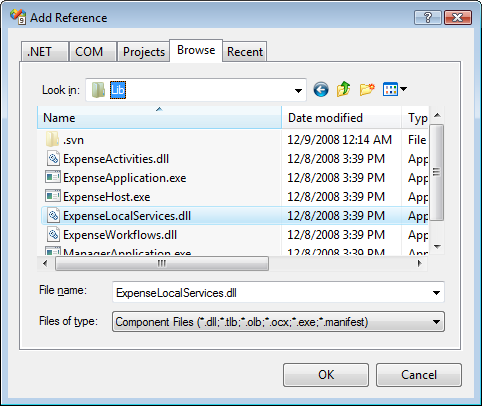
****

Figure : Add Reference Dialog

1. Press **OK**.

## Task 3 – Add the ExpenseReportSubmitted activity

In this task, you will add a special WF activity to the workflow that knows how to work with data sent to the workflow. The **HandleExternalEventActivity** is used to get data from types outside of the workflow and bind the data to objects the workflow understands.

#### Creating 2 workflow fields to hold data values

In the *Workflow1* class we need to create two member variables. One variable will hold a reference to **ReportSubmitted**’s event arguments, the report object. The second variable will be used to hold the expense amount for the report that was submitted.

We’ll use these variables later in our workflow.

#### Adding local fields to the workflow

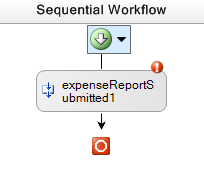
1. Open *Workflow1.xoml.vb* for editing (**Right-click | View | Code** from the visual designer).
2. Add the following lines to the class file.

|  |
| --- |
| **Snippet:** WFLab01\_Ex03\_Task03\_MemberVariables |
| **Public Class Workflow1**  **Inherits SequentialWorkflowActivity**  Public reportArgs As ExpenseLocalServices.ExpenseReportSubmittedEventArgs  Public Amount As Integer  **End Class** |

#### Adding a HandleExternalEventActivity to the workflow

|  |
| --- |
| A Note on HandleExternalEventActivity The **HandleExternalEventActivity** activity is used in conjunction with the **CallExternalMethodActivity** activity for input and output communications with a local service.  The base **HandleExternalEventActivity** class blocks the workflow until the event specified by the **InterfaceType** and **EventName** properties is raised by the corresponding local service registered with the **WorkflowRuntime**. After the event is raised, or if it was raised before the activity started executing, the inbound data is assigned to bound locations as defined in the **ParameterBindings** collection. |

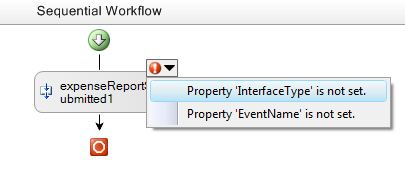
1. Open *Workflow1.xoml* in the Visual Studio workflow designer (double-click the file in **Solution Explorer**).
2. Open the Toolbox (**View | Toolbox** ) to display activities available to add to your workflow.
3. Under the Windows Workflow v3.0 group, select the **HandleExternalEvent** activity.
4. Drag and drop it onto the workflow design surface.
5. Set the **Name** property for the **HandleExternalEvent** activity to **expenseReportSubmitted1** (**Right-click the Activity | Properties**)
6. Your workflow should look like this:

****

**Figure 29: Named Activity**

#### Configuring the HandleEventActivity

1. Click on the red exclamation point (smart tag) on the new activity and select the menu item labeled **Property ‘InterfaceType’ is not set**.

****

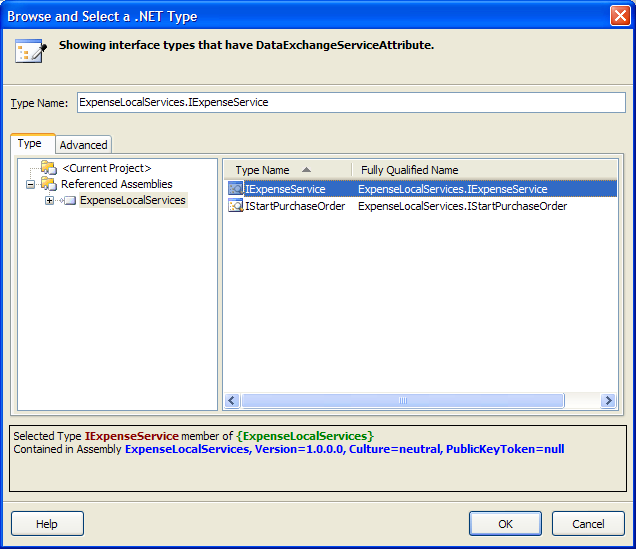
**Figure 30: Using Activity's Smart Tag**

1. The **HandleExternalEvent** **Activity**’s property dialog opens revealing the missing properties. The **HandleExternalEvent Activity** has several properties that must be set in order for it to do its work.

#### Setting the InterfaceType property

1. Click on the **InterfaceType** property. Click the ellipse […] button to show the **.NET Type Browser** dialog.
2. Choose **ExpenseLocalServices.IExpenseService** from the **Referenced Assemblies** tree.

|  |
| --- |
| Choosing the InterfaceType This tells the activity what type it will be talking to when exchanging the data. Whatever concrete object is used; it must implement the .NET interface specified by the **InterfaceType** property. |

****

**Figure 31: Selecting an Interface for the Workflow**

1. Click **OK** to close the **.NET Type Browser** dialog.

#### Setting the EventName property

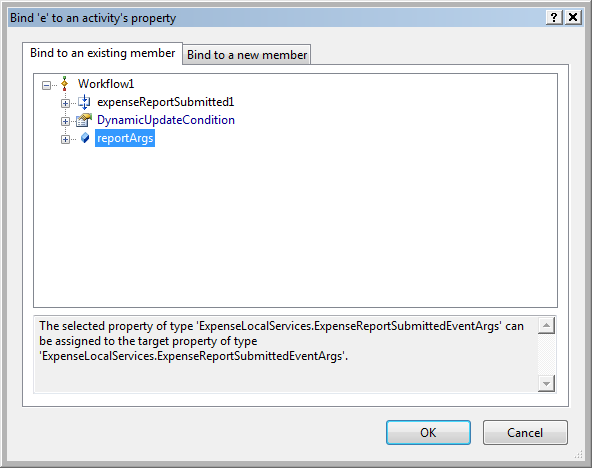
|  |
| --- |
| The **EventName** property specifies which event the activity will subscribe to. The **EventName** property specifies an event declared in the .NET interface mapped in the **InterfaceType** property. |

1. In the properties window for expenseReportSubmitted1, use the drop-down menu for **EventName** to select **ExpenseReportSubmitted**

#### Setting EventArgs

|  |
| --- |
| Using EventArgs The **e** property maps the argument passed via the event to type declared within the workflow. |

1. Click on the **e** property; then click the ellipsis which will bring up the following screen property binding interface:  
     
   This specifies that the EventArg passed into the **ExpenseReportSubmitted** handler will be set to the instance of the **Workflow1.reportArg** object.

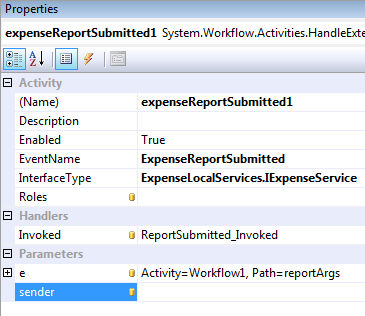
****

**Figure 32: Binding the event argument to a class member**

1. Select **reportArgs** and click **OK**.

#### Setting the Invoked property

1. For the **Invoked** property, enter the value **ReportSubmitted\_Invoked** and press **Enter** to be taken to the code-beside file. Before pressing **Enter**, the properties window should look like this:

****

**Figure 33: Activity properties fully completed**

1. Add code in the **ExpenseReportSubmitted** method to write the method name to the console and set the **amount** variable using the data from the event. The code should read as follows:

|  |
| --- |
| **Snippet:** WFLab01\_Ex03\_Task03\_ReportSubmittedInvoked |
| **Private Sub ReportSubmitted\_Invoked(ByVal sender As System.Object, \_**  **ByVal e As System.Workflow.Activities.ExternalDataEventArgs)**  Console.WriteLine("ReportSubmitted\_Invoked")  Me.Amount = Me.reportArgs.Report.Amount  **End Sub** |

1. Perform a test compile (**CNTL+Shift+B**). The solution should build.

## Task 4 – Add the pre-built AutoApprove activity to the workflow

Now, add a pre-built custom activity that will calculate whether a submitted report can be automatically approved. The **AutoApprove** activity contains logic to approve expenses under $1000. This evaluation results in the activity setting its own **Approved** property to true or false. To use this logic, the **AutoApprove** activity must first be added to the **Toolbox** so it can be used in the designer.

|  |
| --- |
| A Note on Custom WF Activity Re-Use This is a typical technique for sharing and re-using custom WF Activities. Custom WF Activities may come from a vendor, or be developed in-house. |

#### Adding the custom WF Activity to the Toolbox

1. Select **Tools | Choose Toolbox Items…** from the menu.
2. Select the **.NET Framework Components** tab and click the **Browse** button.
3. Navigate to the directory below and open ***ExpenseActivities.dll*** *C:\WF\WF 3.5 Labs\Lab01\resources\Lib*
4. Click **OK** on the **Choose Toolbox Items** form.
5. In the **Toolbox**, inside the **General Group**, you should now seethe**AutoApprove** and **GetManager** custom WFactivities as shown in the figure below.

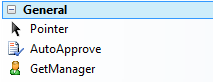


Figure : New Activities in the Toolbox

#### Adding the custom activity to the workflow

1. Open *Workflow1.xoml* in the Visual Designer (double-click the file in **Solution Explorer**).
2. Drag the **AutoApprove** activity onto your workflow after **expenseReportSubmitted1**. Your workflow should now look like the picture below.

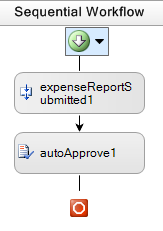


Figure : Current state of the Workflow1

#### Configuring the new AutoApprove activity

1. Open the Properties for **autoApprove1** (**Right-click** the activity **| Properties**)
2. In the **Amount** property for **autoApprove1**, click on the ellipsis to bind the property to **Workflow1.amount**.

## Task 4 – Add a Rule Condition to the workflow

In this task, add a conditional activity to the workflow that allows the workflow to take one of 2 paths of execution, depending on a value passed to the workflow. This conditional logic available in a WF activity: the **IfElse** activity. The **IfElse** activity is a special type of activity known as a **Composite activity**.

|  |
| --- |
| Basic and Composite Activities WF provides for two types of activities, **Basic** and **Composite**. **Basic** activities do work themselves. **Composite** activities host other activities inside them. **Composite activity** functionality is defined by the composition of its child activities.  You may think of this as a control that consists of other child controls. |

Some **Composite activities** have the special job of providing conditional logic to workflows. These activities use a logical construct known as **Rule Conditions**.

|  |
| --- |
| Rule Conditions A Rule Condition is a logical conditional statement stored as XML with the workflow. It can include predicates that compare workflow state and Boolean algebra. Further, multiple predicates may be combined for more complex logic.  Rule conditions can be used in various activities including **IfElse**, **While**, **ConditionedActivityGroup**, and **Replicator**. |

#### Adding an IfElse activity to the workflow

1. Open *Workflow1.xoml* in the Visual Designer (double-click the file in **Solution Explorer**).
2. Add an **IfElse** to the workflow below the **autoApprove1** activity. (Drag an **IfElse** activity from the Toolbox and drop it on the workflow).
3. Set the **Name** property for the **IfElse** activity to **EvaluateExpenseReport**.

#### Configuring the first conditional branch

1. Select the left section of the **IfElse** activity, labeled **ifElseBranchActivity1**.
2. In the properties window enter or select values for the following properties:
3. **Name** – Enter the value **IfAutoApproveReport**
4. **Condition** – Select **Declarative Rule Condition** from the drop down list.
5. Click the plus **[+]** to expand the **Condition** property in the properties window and enter the value **AutoApproveCondition** for the **ConditionName**.
6. Enter the **Expression** property by clicking on the ellipsis to bring up the **Rule Condition Editor**, as shown below.

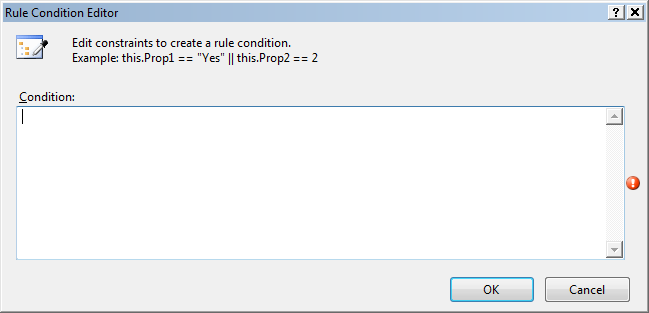


Figure : The Rule Condition Editor

1. Enter this.autoApprove1.Approved in the condition box and click **OK**.   
   **Hint:** Try typing it out. The editor provides IntelliSense.

|  |
| --- |
| Condition Manager Tool If you click on the ellipses next to the Name property you will view a dialog window that allows you to manage the declarative conditions. Try this feature next time you create conditions in workflows.    Figure : Condition Manager Tool Launch Button |

#### Configuring the second conditional branch

1. In the Visual Studio workflow designer, select the right section of the **EvaluateExpenseReport** activity labeled **ifElseBranchActivity2**.
2. In the properties window, set the **Name** property for the branch to the value **ElseRejected.**

## Task 5 – Add the logic to approve or reject the expense report

Now the conditional logic is set up and the workflow has 2 paths of possible execution. If the expense report was approved in the **AutoApprove** activity, the **IfAutoApproveReport** branch will fire. If the expense report is not approved, the **ElseRejected** branch will fire.

In both cases, we want to call out to some code provided in the **ExpenseLocalService** assembly. We call into that assembly using the **CallExternalMethod** activity.

#### Implementing the first conditional branch

1. Select the **CallExternalMethod** activity from the Toolbox and drag and drop it onto the workflow design surface inside of the branch labeled **IfAutoApproveReport**.
2. Set the **Name** property of the new activity to **approveExpenseReport1**.
3. Set the **InterfaceType** property the same as for the **HandleExternalEvent** activity in Task 3. 
   1. Click the ellipses […] next to **InterfaceType** property.
   2. Do this by selecting **ExpenseLocalService.IExpenseService** from the dialog you get when you click the ellipses […] next to **InterfaceType**.
4. Choose the **MethodName** and select **ApproveExpenseReport** from the drop down list.
5. Set the **Report** property by clicking the ellipsis to bind this property to **reportArgs.Report**, as shown in the screen below:

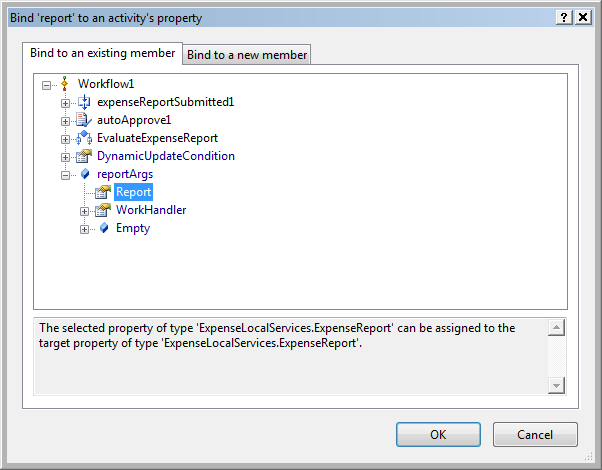


Figure : Binding the Report object to the appropriate activity property

#### Implementing the second conditional branch

1. Add another **CallExternalMethod** activity to the workflow definition by dragging and dropping it into the **ElseRejected** branch of our **IfElse** activity.
2. Set the properties of the new CallExternalMethod activity according to the following table.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | rejectExpenseReport1 |
| **InterfaceType** | ExpenseLocalServices.IExpenseService |
| **MethodName** | RejectExpenseReport |
| **Report** | reportArgs.Report |

#### The workflow Visual Designer should look like this

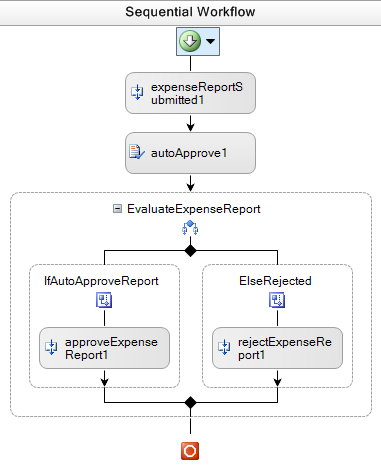


Figure : Current workflow state

## Task 6 – Configure the ExpenseWorkflows project for debugging

The workflow needs a way to be run and debugged. A host application has been provided for this purpose. The host application is a command line executable that listens for messages from client applications. When the host receives a request for processing from a client application, it invokes the workflow from the lab.

#### Adding the host application to the project output folder

1. Using Windows Explorer, copy the files **ExpenseHost.exe** and **ExpenseHost.exe.config**
2. From this folder:

*C:\WF\WF 3.5 Labs\Lab01\resources\Lib*

1. To this folder:

*C:\WF\WF 3.5 Labs\Lab01\VB\before\ExpenseWorkflows\ExpenseWorkflows\bin*

#### Configuring the project to use this host application for debugging

1. Open the **Project Properties** window for the **ExpenseWorkflows** project.

**Right-click** on the **ExpenseWorkflows** project in **Solution Explorer** and select **Properties** from the context menu.

1. The **Project Properties** window is displayed.
2. Click the **Debug** tab in the **Project Properties** window
3. Change the **Start Action** to **Start External Program**
4. Enter the following path or browse to the test host application at the following path:
5. *C:\WF\WF 3.5 Labs\Lab01\VB\before\ExpenseWorkflows\ExpenseWorkflows\bin\ExpenseHost.exe*
6. The Properties window should look like this:

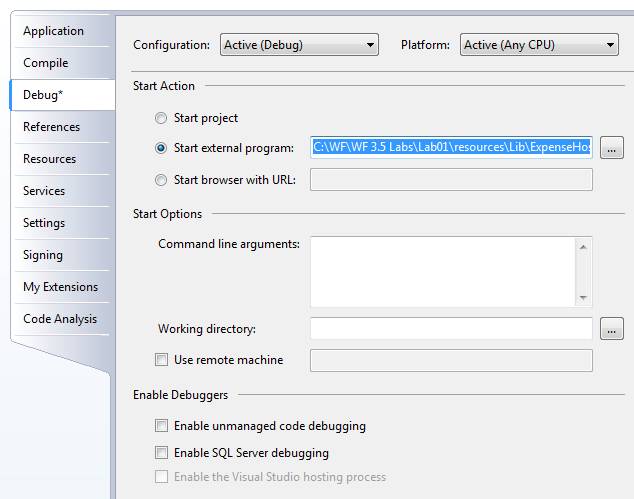


Figure : Project Properties Debug settings

1. Save the settings (**CTRL+S)**.

#### Setting a breakpoint in the workflow

1. Open *Workflow1.xoml* file in the Visual Studio workflow designer (double-click the file in **Solution Explorer**).
2. **Right-click** the **expenseReportSubmitted1** activity and choose **Breakpoint | Insert Breakpoint**.

## Task 7 – Test the Expense Reporting Workflow

#### Running the workflow in the workflow host

1. Compile and run the **ExpenseWorkflows** project (Press **F5** or select **Debug | Start Debugging**).
2. **ExpenseHost.exe** starts and displays a console window. It is waiting for expense reports to be submitted.

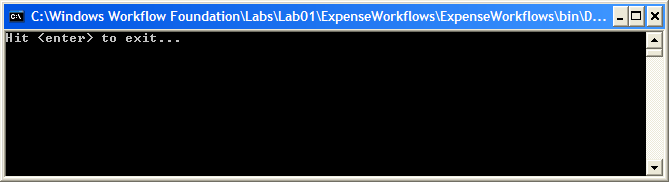


Figure : The workflow is started.

#### Running the WinForm application that calls the workflow

Now that we have a host that can create and run workflows, you need a way to talk to the host and invoke a workflow instance. A Windows Forms application has already been provided for you for this purpose. Running the application will allow you to send expense reports to the workflow.

1. Run the **ExpenseApplication.exe** located in the following directory:

*C:\WF\WF 3.5 Labs\Lab01\resources\Lib*

1. Using the **Expense Application**, click the **Submit Report** button to create a new expense report. You may use the default values on the form and submit it to the workflow.

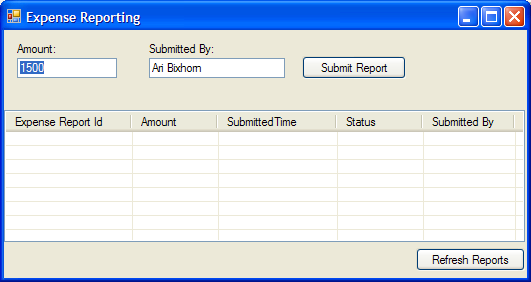


Figure : Expense Report submission application

|  |
| --- |
| An Important Note on Debugging in Visual Studio You may get a failure from Visual Studio while debugging after clicking the **Submit Report** button in the Expense Application. To avoid this, check and set these debugging settings:   * **Tools | Options | Debugging | General** Uncheck the box labeled “Redirect all console output to the Quick Console window”. * **Tools | Options | Debugging | Edit and Continue** Uncheck the box labeled “Enable Edit and Continue”. * **Tools | Options | Debugging | Just-In-Time** Ensure that the boxes labeled “Managed” is checked. |

#### Stepping through the workflow with the debugger

1. Visual Studio 2008 will break into debug mode on the **expenseReportSubmitted1** activity.

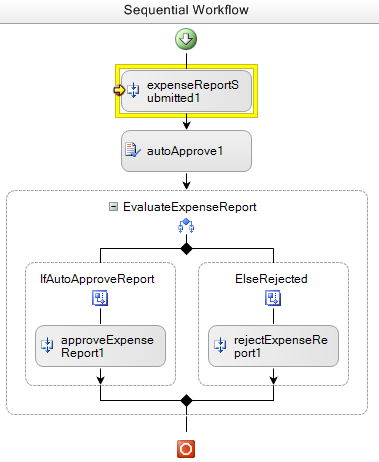


Figure : Debugger stopped on breakpoint

1. Use **F11** (or **Debug** | **Step Into)** to step to the **ReportSubmitted\_Invoked** handler code.

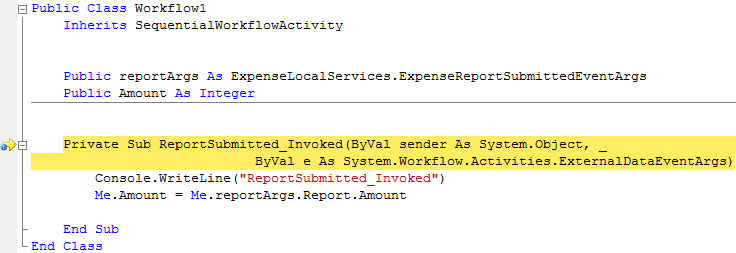


Figure : Debugging the handler

1. Continue stepping through the code and the workflow until you come to the **rejectExpenseReport1** activity as shown below (**F10** or **Debug** | **Step Into**).

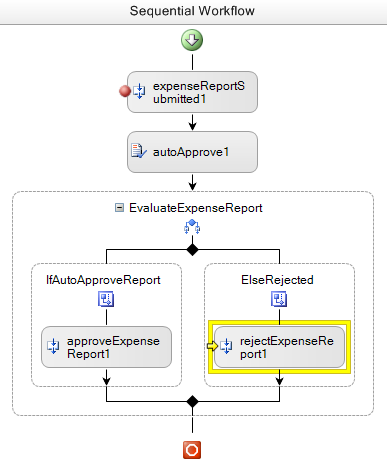


Figure : Debugger walking through the workflow visually

1. Complete running the workflow instance now (**F5** or **Debug | Continue**).

#### Confirming the workflow

1. In the Expense Reporting application, click the **Refresh** button to view the status for the expense report that was just submitted. Notice that the status is rejected because the amount for the expense report was $1500 (by default) and only expense reports less than $1000 are automatically approved.
2. Submit another expense report with an amount of $500 and debug breakpoint hits again. Use **F5** (or **Debug | Continue**) to complete the workflow and notice that the report is automatically approved.

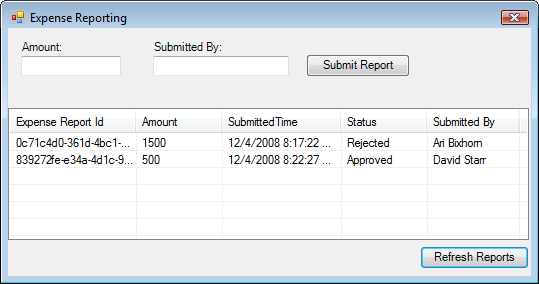


Figure : Expense reporting application

1. The workflow will now complete. You can now close the Expense Reporting application.
2. Stop the **ExpenseHost** console application and stop debugging the **ExpenseWorkflows** project (**Debug | Stop Debugging**).

# Exercise 4 – Using Listen, Delay, and event-based custom activities

This exercise extends the Expense Reporting workflow created in Exercise 3, providing for a manager to manually approve or reject expense reports that are greater than $1000. This demonstrates the ability to include a human-performed action into the workflow.

This exercise includes additional activities including Listen, Delay, and some custom activities.

## Task 1 – Add activities for requesting manager approval

#### Adding new data fields to the base workflow class

1. Open *workflow1.xoml.vb* for editing (**Right-click | View | Code** from the visual designer).
2. Add declarations for ReportEmployeeId, ManagerEmployeeId and reviewArgs to the start of the *Workflow1* class.

|  |
| --- |
| **Snippet:** WFLab01\_Ex04\_Task01\_MemberVariables |
| **Public Class Workflow1**  **Inherits SequentialWorkflowActivity**  Public ReportEmployeeId As String  Public ManagerEmployeeId As String  Public reviewArgs As ExpenseLocalServices.ExpenseReportReviewedEventArgs  **Public reportArgs As ExpenseLocalServices.ExpenseReportSubmittedEventArgs**  **Public Amount As Integer**    **Private Sub ReportSubmitted\_Invoked(ByVal sender As System.Object, \_**  **ByVal e As System.Workflow.Activities.ExternalDataEventArgs)**  **Console.WriteLine("ReportSubmitted\_Invoked")**  **Me.Amount = Me.reportArgs.Report.Amount**  **End Sub**  **End Class** |

#### Adding the GetManager activity

1. Open the *Workflow1.xoml* file in the Visual Studio workflow designer.
2. Delete the **rejectExpenseReport1** activity *(***Right-click** | **Delete**).
3. Change the **Name** property of the **ElseRejected** activity to **ElseManagerApproval** using the properties dialog*.*
4. Add a **GetManager** activity to the **ElseManagerApproval** branch

**Note:** This is a custom activity included from *ExpenseActivities.dll*. It was added to the Toolbox in exercise 3.

#### Binding GetManager properties to new the data fields in the workflow

1. Set the properties on the **GetManager** activity as shown below, setting the activities name and using the ellipses (…) to bind the ID values to the variables you just added.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | getManager1 |
| **ManagerEmployeeId** | Activity=Workflow1, Path=ManagerEmployeeId |
| **ReportEmployeeId** | Activity=Workflow1, Path=ReportEmployeeId |

1. The workflow should now look like the picture below.

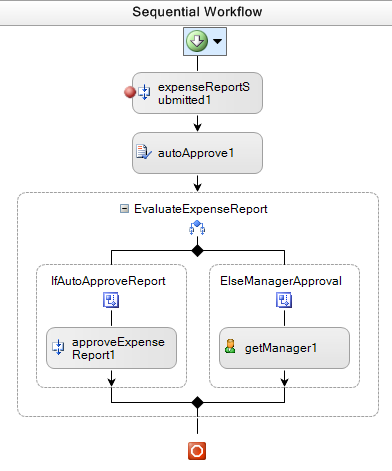


Figure : Current workflow condition

#### Updating the *Workflow1.xoml.vb* code file

1. Open the *Workflow1.xoml.vb* code file for editing (**Right-click | View | Code** from the visual designer).
2. In the **ReportSubmitted\_Invoked** method, set the **ReportEmployeeId** to the value provided in the Expense Report’s **EmployeeId** property.

|  |
| --- |
| **Add the code highlighted in yellow below.** |
| **Private Sub ReportSubmitted\_Invoked(ByVal sender As System.Object, \_**  **ByVal e As System.Workflow.Activities.ExternalDataEventArgs)**  **Console.WriteLine("ReportSubmitted\_Invoked")**  **Me.Amount = Me.reportArgs.Report.Amount**  Me.ReportEmployeeId = Me.reportArgs.Report.EmployeeId  **End Sub** |

This delays the binding until the report is actually submitted. This association cannot be made at runtime.

#### Adding a call to an external method

1. Switch back to the *Workflow1.xoml* file in the Visual Studio workflow designer.
2. Add a **CallExternalMethod** activity to the workflow just below the getManager1activity by dragging and dropping it out of the Toolbox onto the Visual Studio workflow designer.
3. Set the properties on the **CallExternalMethod** activity as shown below, remembering that you can use the ellipses (…) to bind to the existing variables.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | requestManagerApproval1 |
| **InterfaceType** | ExpenseLocalServices.IExpenseService |
| **MethodName** | RequestManagerApproval |
| **ManagerEmployeeId** | Activity=Workflow1, Path=ManagerEmployeeId |
| **Report** | Activity=Workflow1, Path=reviewArgs.Report |

1. The **EvaluateExpenseReport** activity should now look like the picture below (*Note*: the screenshot only shows a portion of the entire workflow).

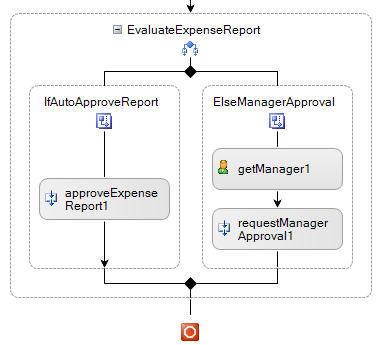


Figure : Current CallExternalMethod activity configuration

## Task 2 – Add a Listen activity for waiting for the Expense Report Review

The **Listen** activity makes the workflow wait for any one of several possible events before the activity proceeds. This workflow will use a Listen activity to wait for the manager to approve the large expense. The Listen activity is often used by WF developers to implement time-out scenarios, as you will implement below.

#### Adding a Listen activity

1. Add a **Listen** activity to the workflow definition below the **requestManagerApproval1** activity by dragging and dropping it out of the Toolbox onto the Visual Studio workflow designer.
2. Change the **Name** property for **listenActivity1** activity to the value **ListenForManagerApproval**.
3. Change the **Name** property for the **eventDrivenActivity1** section of the **Listen** activity to **ManagerReviewed**
4. Change the **Name** property for the **eventDrivenActivity2** section of the Listen activity to **ReviewTimeout**
5. The **Listen** activity should look like this (*Note*: the screenshot only shows a portion of the entire workflow):

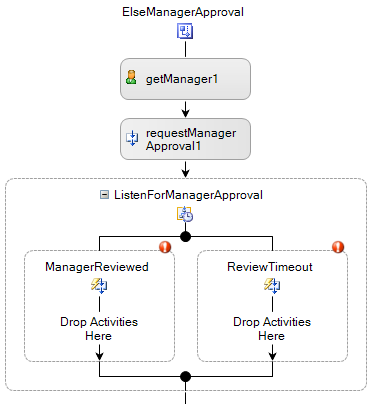


Figure : Current Listen activity configuration

#### Adding a HandleExternalEvent activity that fires when the manager reviews the report and makes a decision

1. Add a **HandleExternalEvent** activity to the workflow definition within the **ManagerReviewed** section by dragging and dropping the activity out of the Toolbox onto the Visual Studio workflow designer.
2. Configure properties for **HandleExternalEvent** activity as shown.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | expenseReportReviewed1 |
| **InterfaceType** | ExpenseLocalServices.IExpenseService |
| **EventName** | ExpenseReportReviewed |
| **e** | Activity=Workflow1,  Path=reviewArgs |

#### Adding a Delay activity in the case the manager does not make a decision in a timely manner

This activity causes the workflow to wait a set amount of time, giving the manager the opportunity to make a decision. This prevents the expense report from being held in approval limbo.

1. Select the **Delay** activity from the Toolbox and drag and drop it onto the workflow design surface within the **ReviewTimeout** branch on the Listen activity.
2. Configure properties for **Delay** activity as shown. For demo purposes, the workflow will only wait 30 seconds before the workflow resumes.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | delayActivity1 |
| **TimeoutDuration** | 00:00:30 |

1. The **Listen** activity should now look like this (*Note*: the screenshot only shows a portion of the entire workflow):

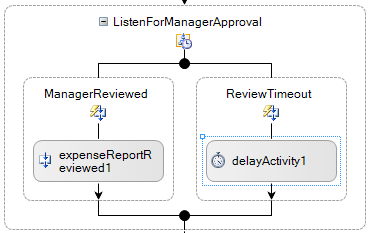


Figure : Current Listen activity configuration

## Task 3 – Approve or Reject the Expense Report

The manager may choose to approve or reject the expense report, so the workflow must take appropriate action in either case.

#### Adding IfElse activity for manager’s decision

1. Add an **IfElse** activity to the workflow definition below the **expenseReportReviewed1** activity.
2. Change the **Name** property to **EvaluateReview**.

#### Configuring the left branch of the IfElse activity

1. Select the left branch of the **EvaluteReview** activity.
2. Set values for the following properties:
   1. **Name** - Enter the value **IfApproved**
   2. **Condition** – Select **CodeCondition** from the drop-down list, indicating that this should be a code-based condition
   3. Expand the **Condition** property, and for the child **Condition** property enter the value **IfReportApproved\_Condition**. Your properties dialog will look like this.

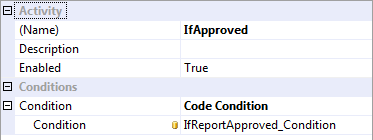


Figure : EvaluateReview branch properties

1. Press enter in the **Condition** property text box and the handler will be added to the code-beside file.
2. In the **IfReportApproved\_Condition** event handler, set the **ConditionalEventArgs.Result** property to the Boolean value stored in the **reviewArgs.Review** object’s **Approved** property as shown below:

|  |
| --- |
| **Private Sub IfReportApproved\_Condition(ByVal sender As System.Object, \_**  **ByVal e As System.Workflow.Activities.ConditionalEventArgs)**  e.Result = Me.reviewArgs.Review.Approved  End Sub |

#### Configuring the right branch of the IfElse activity

1. Open the Visual Studio workflow designer again.
2. Rename the right branch of the **EvaluateReview** activity to **ElseDeclined**.
3. Your **Listen** activity should look now like this (*Note*: the screenshot only shows a portion of the entire workflow):

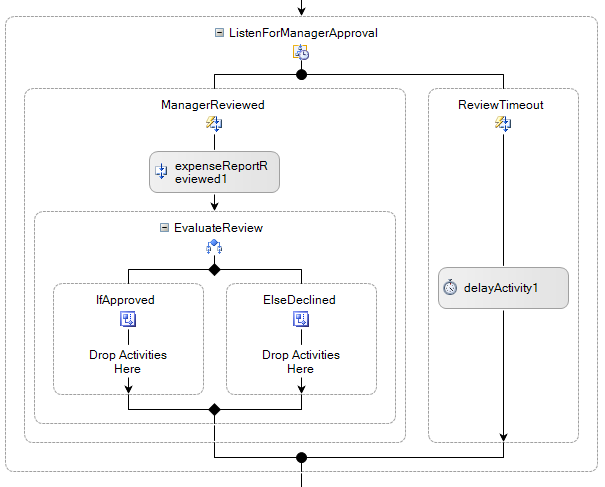


Figure : Current Listen activity configuration

#### Adding logic in case the manager approves the expense

The left condition branch is fired if the manager approves an expense. Now the action to be taken by the workflow must be added.

1. Add a **CallExternalMethod** activity to the workflow definition within the **IfApproved** branch of the **IfElse** activity.
2. Configure the properties of the **CallExternalMethod** activity as shown.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | approveExpenseReport2 |
| **InterfaceType** | ExpenseLocalServices.IExpenseService |
| **MethodName** | ApproveExpenseReport |
| **report** | Activity=Workflow1, Path=reportArgs.Report |

1. The **Listen** activity should look like this (*Note*: the screenshot only shows a portion of the entire workflow):

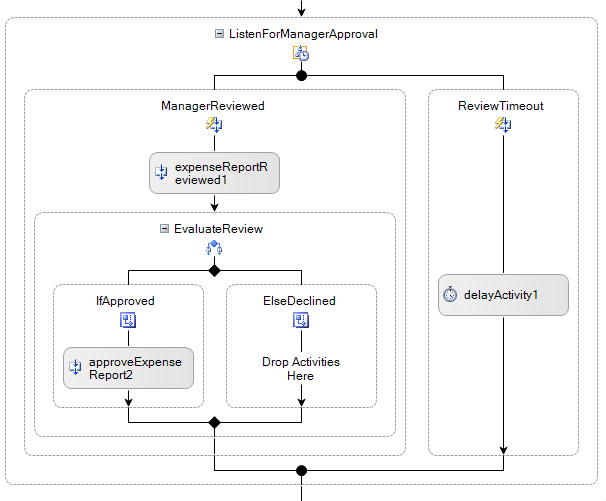


Figure : Current Listen activity configuration

#### Adding logic in case the manager rejects the expense

1. Add a **CallExternalMethod** activity to the workflow definition within the **ElseDeclined** branch of the **IfElse** activity.
2. Configure the properties of the **CallExternalMethod** activity as shown.

|  |  |
| --- | --- |
| Property | Value |
| **Name** | rejectExpenseReport1 |
| **InterfaceType** | ExpenseLocalServices.IExpenseService |
| **MethodName** | RejectExpenseReport |
| **Report** | Activity=Workflow1, Path=reportArgs.Report |

1. The **Listen** activity should look like this (*Note*: the screenshot only shows a portion of the entire workflow):

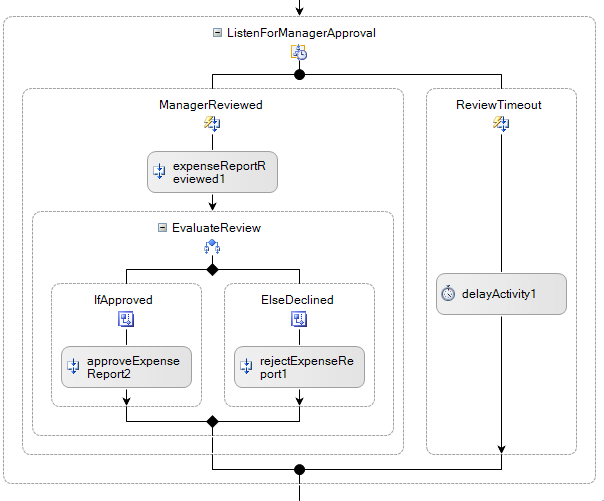
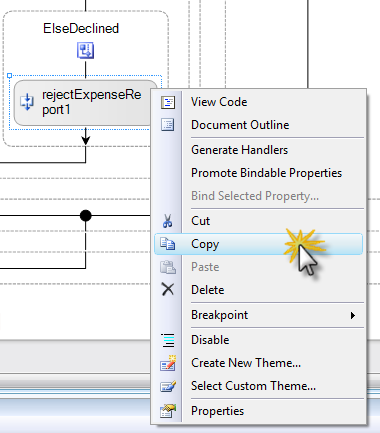


Figure : Current state of Listen activity

#### Adding logic in case the manager never decides

If the manager does not make a decision within 30 seconds, the workflow will go ahead and process. You can reuse the logic already created in case the manager chooses to reject the expense.

1. **Right-click** on the **rejectExpenseReport1** activity and select **Copy** from the context menu.



1. **Right-click** on the Visual Studio workflow designer below the *delayActivity1* activity and select **Paste** from the context menu.
2. Set the **Name** of the pasted activity to **rejectExpenseReport2**.
3. The **Listener** activity should now look like this (*Note*: the screenshot only shows a portion of the entire workflow):

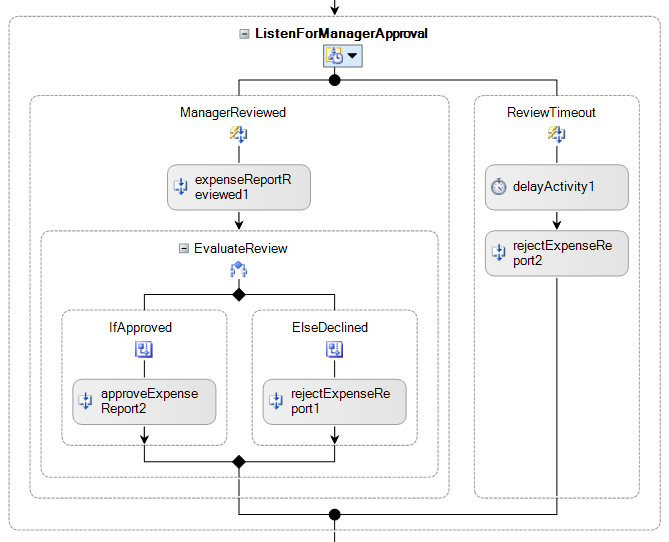


Figure : Current state of Listener activity

1. Your Expense Reporting Workflow is now complete. Your completed workflow should look like the one in the following screenshot:

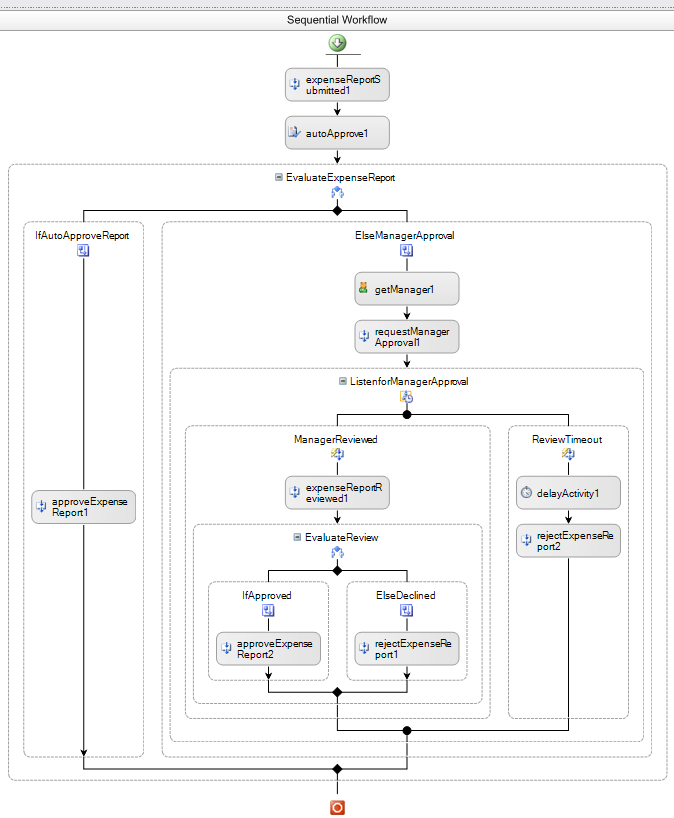


Figure : The completed workflow

## Task 4 – Test the Expense Reporting Workflow

Now that the workflow is created, it’s time to run it and try it out.

#### Verifying that the startup application is correct

1. Navigate to the **Project | ExpenseWorkflows** **Properties** menu.
2. Click on the **Debug** tab.
3. Ensure that the **Start external program** radio button is checked and the program is listed as:  
   *C:\WF\WF 3.5 Labs\Lab01\VB\before\ExpenseWorkflows\ExpenseWorkflows\bin\ExpenseHost.exe*

#### Setting a breakpoint on the workflow

1. Open *Workflow1* the Visual Studio workflow designer.
2. Set a breakpoint at the start of the workflow on the **expenseReportSubmitted1** activity. The top of the workflow should look like this:

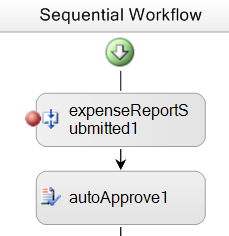


Figure : Breakpoint set on entry activity

#### Running the workflow

1. Compile and run the **ExpenseWorkflows** project under the Visual Studio debugger (**F5** or **Debug | Start Debugging**).
2. The **ExpenseHost** executable will start and display a console window.
3. Run **ExpenseApplication.exe** from the following directory:   
   *C:\WF\WF 3.5 Labs\Lab01\resources\Lib*
4. Run **ManagerApplication.exe** from the following directory:   
   *C:\WF\WF 3.5 Labs\Lab01\resources\Lib*

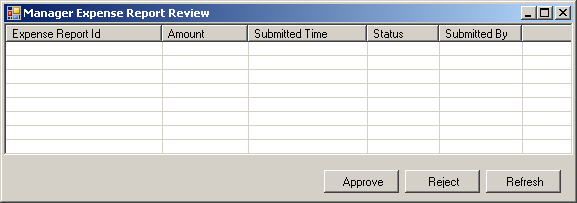


Figure : Manager's application

1. Take a moment to organize your windows so you can see everything happening in a logical order.

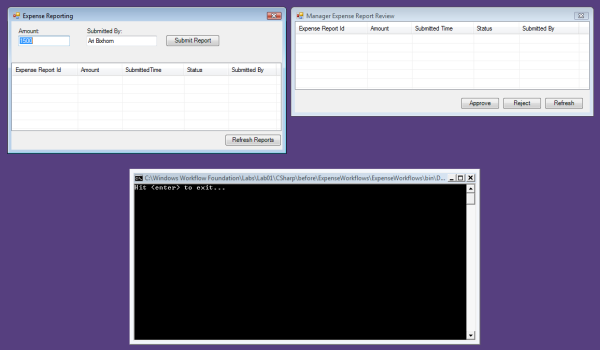


Figure : Preparing to run the workflow

1. Using the **Expense Application**, click the **Submit Report** button to create a new expense report and submit it to the workflow with an amount over $1000. A new workflow instance will be started and Visual Studio 2008 will break into debug mode at the **expenseReportSubmitted1** activity.

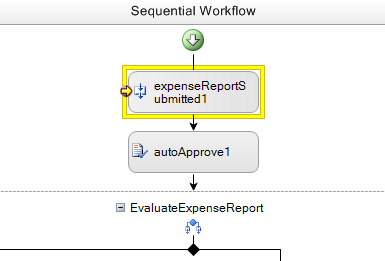


Figure : Workflow stopped in debugger

1. Step through the workflow if you wish (**F10**) or continue the workflow using **F5** or **Debug | Continue**.
2. Now delete all breakpoints using **Debug | Delete All Breakpoints** and then press **F5** to continue.

Now you can submit multiple expense reports of different values.

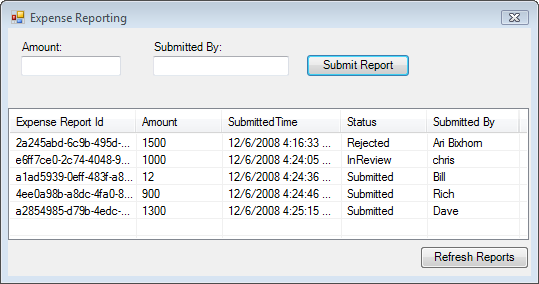


Figure : Several expense reports submitted for approval

1. Using the **Manager** application, click the **Refresh** button to refresh the list of expense reports.
2. Select an expense report from the list and click the **Approve** button to approve the expense report.

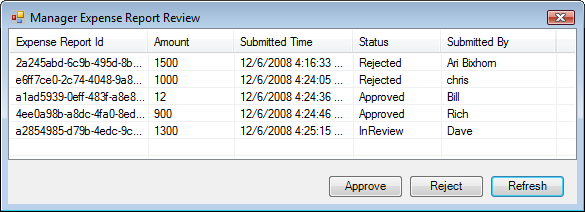


Figure : Several expenses in the process of being managed

1. Finally, using in the **Expense Reporting** application, click the **Refresh** button to refresh the changes.
2. Keep in mind that if you wait for longer than 30 seconds after submitting the expense report, it will automatically be rejected because of the delay activity.
3. Congratulations. You completed this lab.

# Lab Summary

This lab demonstrated several concepts about working with Windows Workflow Foundation.

Create a Hello World workflow

Receiving data into a workflow using parameters

Using activating events, IfElse activity, and declarative conditions

Using Listen, Delay, and custom event-based activities

You authored sequential workflows using the Visual Studio workflow designer. You used several of the basic activities provided by WF as well as some custom activities for receiving events into a workflow and interacting with the host application.

Finally, you saw how workflows can be hosted in a variety of applications including console applications and Windows Forms applications.