

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 02 – Creating Custom Activities 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.

© 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 1](#_Toc218881204)

[Objectives 1](#_Toc218881205)

[System Requirements 2](#_Toc218881206)

[Setup 2](#_Toc218881207)

[Physical Folder Structure 2](#_Toc218881208)

[Code Snippets 2](#_Toc218881209)

[Starting Material 2](#_Toc218881210)

[Acronyms Used in this Lab 2](#_Toc218881211)

[Scenario 3](#_Toc218881212)

[Exercise 1 – Creating a Composite Activity Using the Activity Designer 4](#_Toc218881213)

[Task 1 – Create a new Workflow Project 4](#_Toc218881214)

[Task 2 – Creating an Activity Project 5](#_Toc218881215)

[Task 3 – Finishing the Workflow Solution 11](#_Toc218881216)

[Exercise 2 – Creating a Basic Activity 13](#_Toc218881217)

[Task 1 – Creating a Workflow Project 13](#_Toc218881218)

[Task 2 – Adding an Activity Project 13](#_Toc218881219)

[Task 3 – Using the new custom activity 19](#_Toc218881220)

[Task 4 – Adding an Executor Component 22](#_Toc218881221)

[Task 5 – Building the Project 22](#_Toc218881222)

[Task 6 – Updating the Activity to Send Email 23](#_Toc218881223)

[Exercise 3 – Adding Validation to Check E-Mail Parameters 26](#_Toc218881224)

[Task 1 – Adding an Email Validation Code Activity 26](#_Toc218881225)

[Task 2 – Testing The Validator 29](#_Toc218881226)

[Exercise 4 – Adding an Activity Designer 31](#_Toc218881227)

[Task 1 – Add the Activity Designer 31](#_Toc218881228)

[Task 2 – Testing the Activity Designer 34](#_Toc218881229)

[Lab Summary 37](#_Toc218881230)

# Overview

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

Activities are the unit of execution, re-use, and composition of workflow models. There are two types of WF activities: **basic** and **composite**. Basic activities are steps within a workflow and composite activities can contain other activities.

A base activity library comes with Windows Workflow Foundation and these activities can be used by dragging from the Visual Studio toolbox. Software developers can create their own activities which also appear in the Visual Studio toolbox. Custom activities have the same fidelity as the activities in the base activity library.

Activities are classes and with properties, methods and events which can be referenced from the workflow that hosts the activity. You can build an activity or multiple activities as a .NET assembly and deploy them as re-usable libraries.

## Objectives

The objective of this lab is to demonstrate how to create custom activities in Windows Workflow Foundation. After completing this lab, you will be able to:

* Create custom WF activities
* Know how to expose properties on activities in workflows and access the property values at runtime.
* Provide a rich design-time experience for developers using activities you wrote.

The lab consists of four exercises.

1. In the first exercise, you will create a composite activity using Visual Studio’s built-in activity designer.
2. Next you create a basic activity that we will use to send email.
3. Then you will add a validation component to the workflow to validate the e‑mail parameters.
4. Finally you will learn how to add a custom activity designer to your basic activity to provide a customized interface.

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 lab package.

### 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

Various simplistic scenarios are used in these exercises, which focus on fundamental mechanics of using WF. Scenarios in these exercises qualify primarily as “Hello World” examples.

# Exercise 1 – Creating a Composite Activity Using the Activity Designer

In this exercise, you will create an activity that consists of two separate code activities. You will use the resulting composite activity in a workflow project, running the project to see the output produced by the code activities.

## Task 1 – Create a new Workflow Project

Workflow projects require a host to execute the workflow. For this exercise, you will create a console application to host and interact with your workflow process.

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

Visual Studio displays the **New Project** dialog box.

1. Selectthe **Visual C# | Workflow** project type.
2. Select the **Sequential** **Workflow Console Application** template.
3. The **New Project dialog** should be filled out as:

**Name:** *CompositeActivitySample*

**Path:** *C:\WF\WF 3.5 Labs\Lab02\CSharp\before*

1. Click **OK**.

Visual Studio creates a Workflow console application and opens the Visual Studio workflow designer displaying a graphical representation of the workflow.

1. Open *Program.cs* for editing (double-click the file in **Solution Explorer**).

Visual Studio displays *Program.cs* in the main window.

1. *Program.cs* contains the Main() method for your workflow application. It contains boilerplate code to start the workflow runtime, load and execute the workflow, and then wait for the workflow to complete.
2. In the **Solution Explorer**, right-click on *Workflow1.cs* and rename it to *HelloWorldWorkflow.cs***.**
3. If the Visual Studio displays the following dialog, click “**Yes**”.

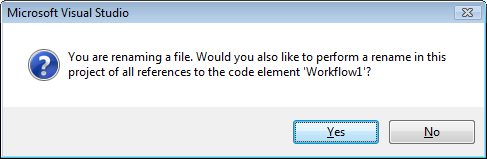


Figure : Confirmation dialog

## Task 2 – Creating an Activity Project

Now that you have created a Workflow project, the next step is to add an activity project to the solution. Once you have created the activity project, you will add two code activities to the activity project. The code activities will execute one after another to demonstrate sequential execution of business process elements. At a very high level, it can be said that there are two types of activities:

* Basic
* Composite

In other words, new activities can be defined either by creating new basic activities where they are implemented from scratch or, they can be composed of other existing activities as composite activities. The activity we will be creating in this exercise will be composing of two code activities, so it falls into the category of “composite” activities.

#### Creating a new Workflow Activity Library project

1. Select the **File | New | Project** menu command.
2. Visual Studio displays the **New Project** dialog box.
3. Select the **Visual C# | Workflow** project type.
4. Select the **Workflow Activity Library** template.
5. Set the project **Name** to *HelloWorldActivityLibrary.*
6. In the dropdown for **Solution**, select **Add to Solution**. This ensures the new project is created as part of the current Visual Studio solution.
7. Select **Add to Solution** from the **Solution** dropdown list.
8. Click **OK**.

Visual Studio creates the activity library project, including a single activity, *Activity1*, and displaysa design view of*Activity1.cs*.

#### Configuring the custom activity

1. Rename *Activity1.cs* to *HelloWorldActivity.cs* by (right click on *Activity1.cs* and choose rename)
2. If the Visual Studio displays the following dialog, click “**Yes**”

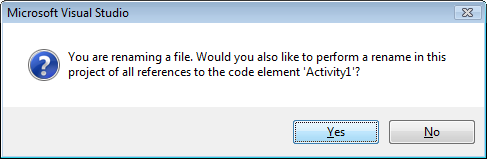


Figure : Rename confirmation dialog

1. Your designer should now have *HelloWorldActivity* loaded as shown in Figure 3, if not double click on *HelloWorldActivity.cs* in the **Solution Explorer**.

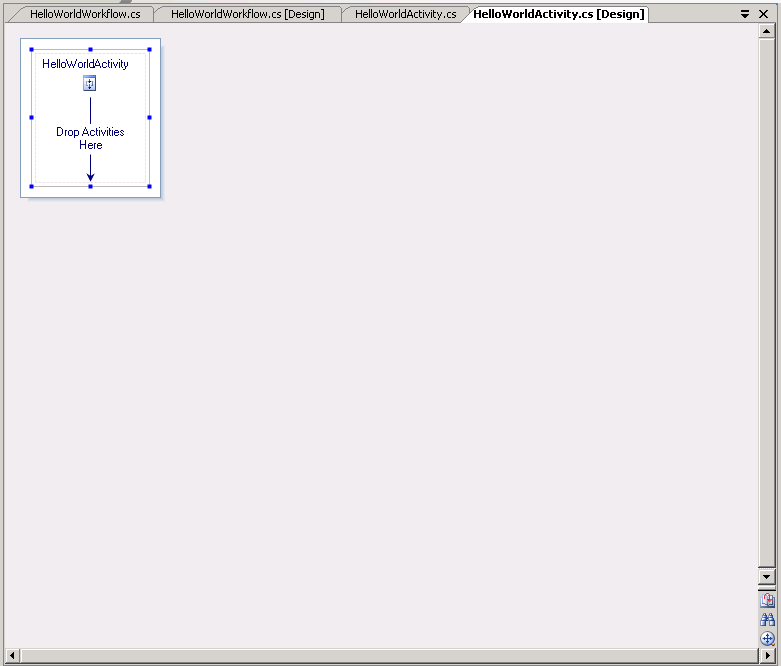


Figure : Visual activity designer

1. Click on the*HelloWorldActivity*iconor surrounding work area to view the properties for *HelloWorldActivity* in the **Properties** window.

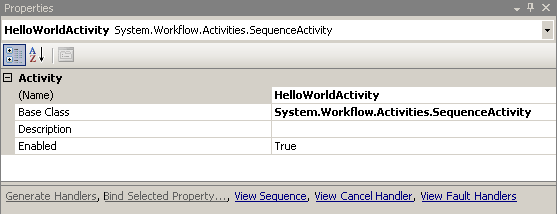


Figure : HelloWorldActivity property window

1. Select the **View | Toolbox** menu command to open the Toolbox window.

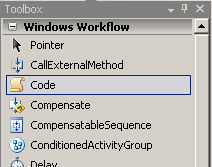


Figure : Visual Studio Toolbox

1. From the **Toolbox** window, drag a **Code** activity (see Figure 5) and drop it onto the “**Drop Activities Here**” work area of *HelloWorldActivity* ‘s design view.
2. The **Activity Designer** displays an activity with the code icon in the work area as shown in Figure 6.

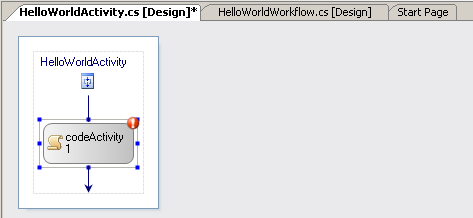
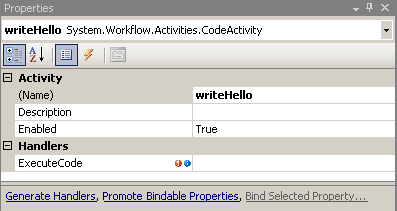


Figure : Code activity in the visual designer

1. In the Property window, change the **(Name)** property of the code activity to *writeHello*.



1. Double click the *writeHello* activity to view the writeHello\_ExecuteCode method.
2. Paste the following code in the writeHello\_ExecuteCode method body.

|  |
| --- |
| A Note on Lab Structure Code for the lab is shown in highlighted regions like the one below. New code added in the step is shown highlighted in yellow. |

|  |
| --- |
| **private void writeHello\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.Write("Hello, ");  **}** |

1. Double click *HelloWorldActivity.cs* in the **Solution Explorer** to return to the Visual Studio workflow designer.
2. Drag a second **Code** activity onto the design area below the first code activity.
3. Change the **(Name)** property to *writeWorld*.
4. Double click the activity with the **Code** icon you just placed onto the design area to open the code view of the writeWorld \_ExecuteCode method.
5. Add the following code in the writeWorld \_ExecuteCode method body.

|  |
| --- |
| **private void writeWorld\_ExecuteCode(object sender, EventArgs e)**  **{**  Console.WriteLine("World.");  } |

1. The code for the *HelloWorldActivity* class should now look like Figure 7.
2. Build the solution to ensure everything is configured properly at this point (**CNTL + Shift + B**).

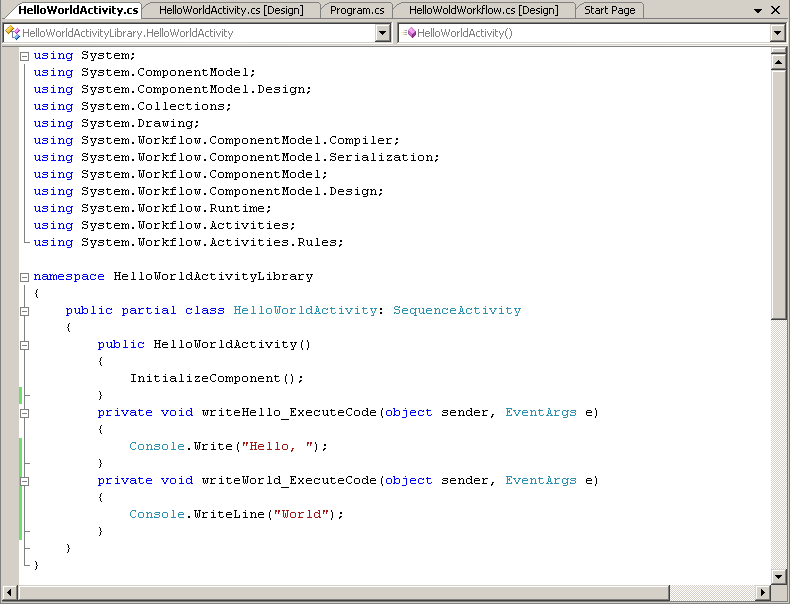


Figure : The HelloWorld activity

## Task 3 – Finishing the Workflow Solution

The **HelloWorld** activity is now a reusable workflow activity. It can be used in the composition of a workflow and that is the focus of this task.

#### Adding the new activity to the workflow

1. In the **Solution Explorer**,double click on *HelloWorldWorkflow.cs* to switch to the **Design** view of the **Workflow**.
2. Open the **Visual Studio Toolbox**  
   Notice that there is a new entry in the **Toolbox** list, **HelloWorldActivityLibrary Components**, under which the activity created in the last Task, *HelloWorldActivity,* appears. See Figure 8.

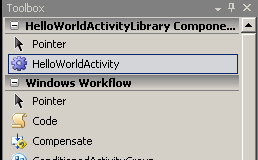


Figure : The re-usable HelloWorldActivity

1. Drag *HelloWorldActivity* onto the Visual Studio workflow designer and drop it onto the area labelled “**Drop Activities to create Sequential Workflow**”. Change the **(Name)** property of the activity to *helloWorld*.
2. The Visual Studio workflow designer updates the display to include *HelloWorldActivity*, including the two code activities, as shown in Figure 9.

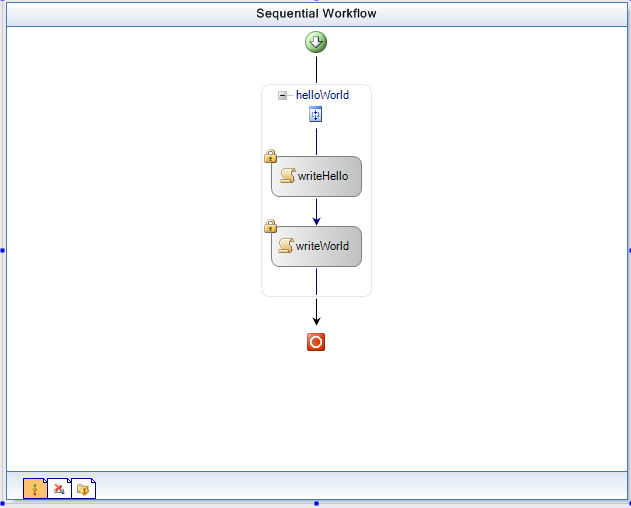


Figure : The visual workflow designer

#### Running the workflow

1. Select the **Build | Build Solution** menu command.
2. Select the **Debug | Start Without Debugging** menu command to run the program and see the following output in the console.

**CropperCapture[43]**

Figure : Output from the activity

1. Press the **Enter** key to end the program.
2. Select **File | Save All**.
3. Select **File | Close Solution**.

# Exercise 2 – Creating a Basic Activity

In this exercise, you will create another Workflow project that contains an activity with custom properties. The custom properties specify fields for an e-mail message.

You will create a basic custom activity and add it to the Workflow. The component will use .NET APIs and the data contained in the fields to send an e-mail message.

## Task 1 – Creating a Workflow Project

1. Select the **File | New | Project** menu command.

Visual Studio displays the **New Project** dialog box.

1. Selectthe **Visual C# | Workflow | Sequential** **Workflow Console Application** project template.
2. Enter the path *C:\WF\WF 3.5 Labs\Lab02\CSharp\before* in the **Location** field.
3. Enter the name *CustomPropertySample* in the **Name** field.

**Name:** *CustomPropertySample*

**Path:**  *C:\WF\WF 3.5 Labs\Lab02\CSharp\before*

1. Click **OK.**

Visual Studio creates a workflow console application and opens the Visual Studio workflow designer, displaying a graphical representation of the workflow.

1. Rename the *Workflow1.cs* workflow to *SendMailWorkflow.cs* by right clicking on *Workflow1.cs* and choosing rename.

## Task 2 – Adding an Activity Project

#### Adding the new project

1. Select the **File | New | Project** menu command.

Visual Studio displays the **New Project** dialog box.

1. Selectthe **Visual C# | Workflow |** **Workflow** **Activity Library** project template.
2. Enter the name *SendMailActivityLibrary* in the **Name** field.
3. Select **Add to Solution** from the **Solution** dropdown list.
4. Click **OK**.

Visual Studio creates the activity library project, including a single activity, *Activity1*, and displaysa graphical design view of*Activity1*.

1. Rename *Activity1.cs* to *SendMailActivity.cs* by right clicking on *Activity1.cs* and choosing rename**.**
2. Click **Yes** on the confirmation dialog.

#### Configuring the new activity

1. Click on *SendMailActivity* title just above the **“Drop Activities Here”** work area to view the properties for *SendMailActivity* in the **Properties** window.

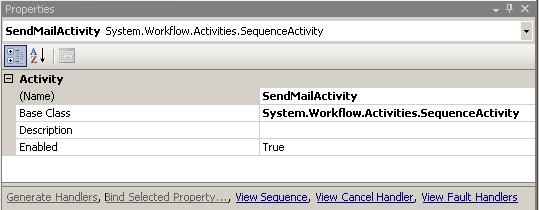


Figure : Activity properties dialog

1. In the **Properties** window, select the **Base Class** property.
2. Click the ellipse **“…”** button associated with the **Base Class** property.
3. The “**Browse and Select a .NET Type”** dialog box appears to allow you to select a base class for this activity.

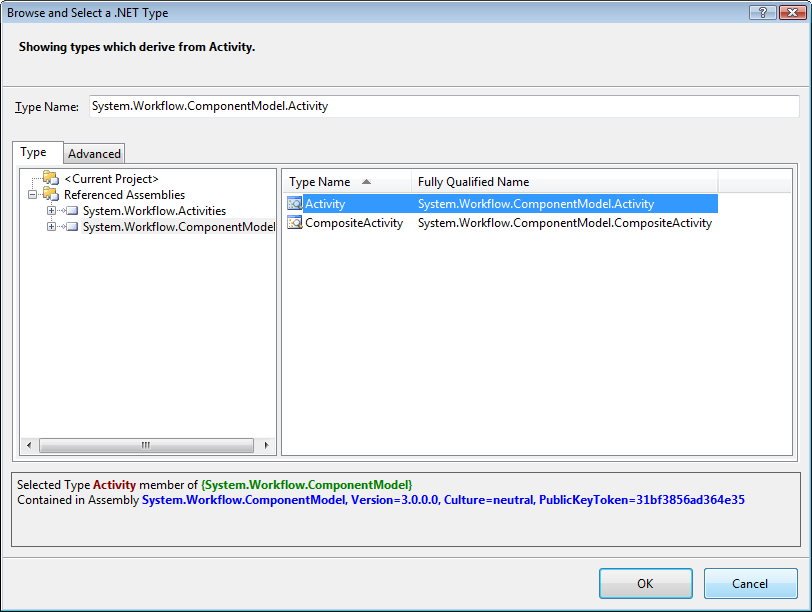


Figure : Base Type selection dialog

1. In the Type tab, select **System.Workflow.ComponentModel**.

The list box will be updated to display the two available base classes under the **System.Workflow.ComponentModel** namespace: **Activity** and **Composite Activity**.

1. Select **Activity** as shown in Figure 12.
2. Click **OK**.

#### Working with dependency properties

1. In the **Solution Explorer**, right click on*SendMailActivity.cs* to bring up the context menu.
2. Select **View Code**.
3. Place the cursor on the closing bracket of the *SendMailActivity* constructor and press Enter **Twice**.

|  |
| --- |
| A note on code snippets Visual Studio includes many code snippets by default. These are in addition to the code snippets installed as part of these labs. The following steps use a code snippet included with Visual Studio and it is available outside of these labs. |

1. At the current cursor location in the code editor:

**Right Click | Insert Snippet.| Other | Workflow | DependencyProperty – Property**

See the following pictures for examples.

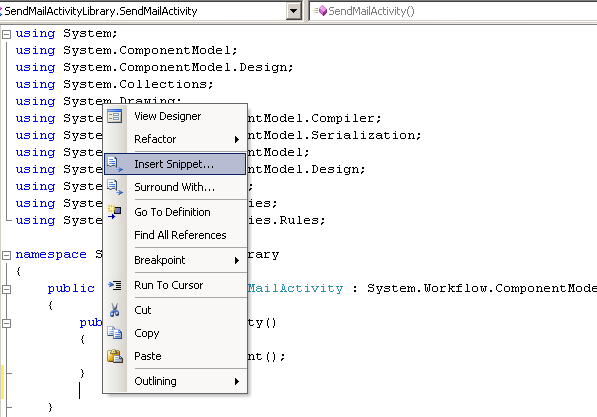


Figure : Inserting a code snippet, 1

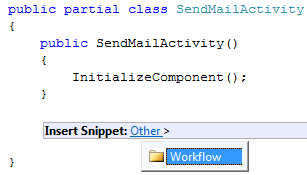


Figure : Inserting a code snippet, 2

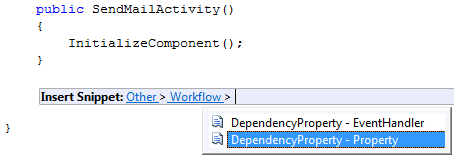


Figure : Inserting a code snippet, 3

1. The snippet adds some boilerplate code to be customized below.
2. Restructure the property declaration as follows:
   1. Name the variable **ToProperty**.
   2. Pass the string “**To**” for the name variable to the constructor. This ties the dynamic property member to the property that will also be named “**To**”.

See the following code for an example.

|  |
| --- |
| **public static DependencyProperty ToProperty =**  **DependencyProperty.Register( "To",**  **typeof(string),**  **typeof(SendMailActivity));** |

1. The following table describes some of some of the attributes that can be added to the dependency property:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| **Browseable** | Boolean | Indicates whether this property appears in the Properties window. |
| **Category** | String | A user-defined category for the property. |
| **Description** | String | A description for the property. |
| **DesignerSerializationVisibility** | **Visible**, **Hidden**, or **Content** | Determines how and if properties will be serialized.  **Visible** (the default) – the property will be serialized normally.  **Hidden** – prevents property serialization  **Content** – used for collection properties. The collection object itself is not serialized, however the contents of the collection are.  If a developer chooses a collection type, this property will be set to **Content**. If a developer chooses a non-serializable type, this property will be set to **Hidden**. |
| **ValidationVisibility** | **Optional**, **Required**, or **Hidden** | Specifies how the property’s value is validated.  **Optional** (the default) – the property can accept null values.  **Required** – The property must be set to a non-null value and is checked to ensure that this is the case.  **Hidden** – There is no automatic validation of the property’s value.  If ValidationVisibility is set to ‘Required,’ when the component is reused, it will require the property to be configured via smart tags, obviating the need for a check in the activity’s Validator class. |

1. Name the property **To**.
2. Fill in these attributes from the boilerplate code with the corresponding values.
   1. **DescriptionAttribute:** To Email Address**.**
   2. **CategoryAttribute:** EmailActivity (this will create its own area in the property designer).

See the following code for an example. Customized code is shown in yellow.

|  |
| --- |
| **[DescriptionAttribute("To Email Address")]**  **[CategoryAttribute("EmailActivity")]**  **[BrowsableAttribute(true)] [DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Visible)]**  **public string To**  **{**  **get**  **{**  **return ((string)(base.GetValue(SendMailActivity.ToProperty)));**  **}**  **set**  **{**  **base.SetValue(SendMailActivity.ToProperty, value);**  **}**  **}** |

1. In the same way, repeating steps 18 through 22, add the following three properties, placing the correct values for each in the appropriate fields:

|  |  |  |  |
| --- | --- | --- | --- |
| Member Variable Name | Name Constructor Parameter | Description Attribute | Category Attribute |
| FromProperty | From | From Email Address | EmailActivity |
| SubjectProperty | Subject | Subject of Email | EmailActivity |
| BodyProperty | Body | Body of Email | EmailActivity |

The activity now has four custom properties.

1. Select the **Build | Build Solution** menu command.

This step will add *SendMailActivity* to the **Toolbox**.

## Task 3 – Using the new custom activity

Now the activity is available as a reusable component for inclusion into a workflow. This task uses the activity in the previously created workflow.

1. Double click on *SendMailWorkflow.cs* in the **Solution Explorer** to switch to design view for the workflow.
2. Notice that the *SendMailActivity* is now present in the **Toolbox** window.

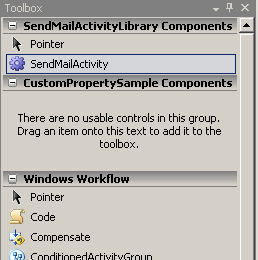


Figure : SendMail Activity in the Toolbox

1. Drag *SendMailActivity* from the Toolbox window and drop it onto the Visual Studio workflow designer area labelled **“Drop Activities to create Sequential Workflow”**.
2. Change the **(Name)** property of the activity to *sendMail*.
3. Notice that the custom properties you created previously are now available in the **Properties Window** for this Activity. See Figure 17.

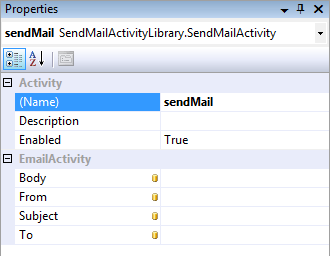


Figure : SendMailActivity properties

#### Setting custom activity properties

You may now set values for the custom properties, allowing you to initialize input values for this activity in your workflow process.

1. Set **Body** to “*body of email*”
2. Set **From** to [*email@any.domain*](mailto:email@any.domain)
3. Set **Subject** to “*subject of email”*
4. Set **To** to *email@some.domain*

The format of the addresses does not matter at this point, as we will not be sending out an email yet. We will be using an e-mail address that is valid for the local SMTP server as we move through the lab.

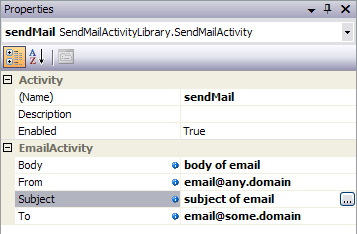


Figure : Custom activity properties

## Task 4 – Adding an Executor Component

The entry point for a WF activity is the Activity’s Execute method. Override this method from the base Activity class to add functionality to your custom Activity.

1. Right click on the *SendMailActivity.cs* file and choose **View Code**.
2. At the bottom of the code file, within the Activity class definition, override the **Execute method** on the activity. Inside this executor method, add the following code that accesses one of the custom properties of the activity and displays its contents.

|  |
| --- |
| **Snippet:** WFLab02\_Ex02\_Task04\_Execute |
| **protected override ActivityExecutionStatus**  **Execute(ActivityExecutionContext executionContext)**  **{**  **Console.WriteLine(To);**  **return ActivityExecutionStatus.Closed;**  **}** |

1. Select the **Build | Rebuild Solution** menu command and ensure the solution builds.

## Task 5 – Building the Project

1. In the **Solution Explorer**, double-click on *SendMailWorkflow.cs* to switch to the Visual Studio workflow designer.
2. Select **Debug | Start Without Debugging** to run the project and see the output on the console.
3. The string you entered previously for the value of the **To** field is written to the console.

CropperCapture[52]

Figure : Workflow application output

1. Press the **Enter** key to end the program.  
   The new custom activity was executed by the workflow you created. One of the custom dependency properties you created was used in the execution.

## Task 6 – Updating the Activity to Send Email

Now that the executor has been added successfully, you will replace the line of code that prints the value of the **To** property to the console with code that actually reads the custom properties and sends e-mail.

|  |
| --- |
| Local SMTP Server This task assumes an SMTP email server is running on the local machine. Set up on the lab computer as described in the readme accompanying these labs. *C:\WF\WF 3.5 Labs\WF - Lab ReadMe and Setup Guide.doc*  If you have troubles with email check the log files in the directory:  c:\windows\System32\LogFiles\SMTPSVC1 |

#### Configuring the activity for email

1. Right click on the *SendMailActivity.cs* file and choose **View Code**.
2. Add the following using directive to the top of the file:

|  |
| --- |
| using System.Net.Mail; |

1. Replace the body of the **Execute** method with the following code that uses the **MailMessage** and **SMTPClient** class to send email:

|  |
| --- |
| **Snippet:** WFLab02\_Ex02\_Task06\_SendMail |
| **protected override ActivityExecutionStatus**  **Execute(ActivityExecutionContext executionContext)**  **{**  MailAddress toAddress = new MailAddress(To);  MailAddress fromAddress = new MailAddress(From);  MailAddressCollection addresses = new MailAddressCollection();  addresses.Add(toAddress);  MailMessage msg = new MailMessage(fromAddress, toAddress);  msg.Subject = Subject;  msg.Body = Body;  SmtpClient mail = new SmtpClient("localhost");  mail.Send(msg);  return ActivityExecutionStatus.Closed;  **}** |

1. Double click on *SendMailWorkflow.cs* to view the Visual Workflow Designer.
2. On the *sendMail* Activity set the **(To)** property to be *user@<your-machine-name>* where <your-machine-name> is obtainable from the readme accompanying these labs.
3. Use **Windows Explorer** to navigate to the directory below so we can watch for our email being delivered.

*c:\inetpub\mailroot\drop*

#### Running the workflow

1. Switch back to **Microsoft Visual Studio**.
2. Select the **Build | Build Solution** menu command.
3. Select **Debug | Start Without Debugging** to run the project.
4. The program will run to completion, sending e-mail using the SMTP client.
5. Switch to **Windows Explorer** to check for your email, you should see a new file.

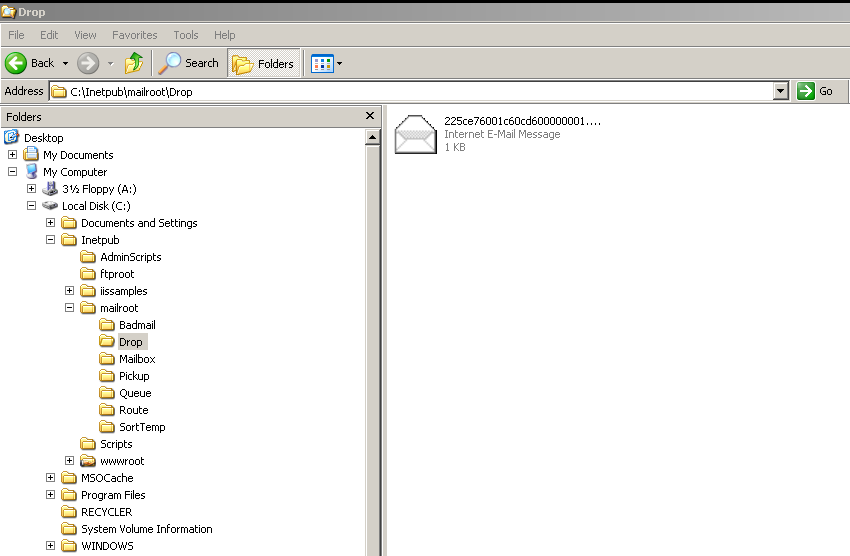


Figure : Email sent to dropbox

# Exercise 3 – Adding Validation to Check E-Mail Parameters

Functionality can be added to custom activities to validate its properties at design time. This functionality can be immensely helpful for workflow developers while working with the activity in the workflow designer.

In this exercise, you will extend the project from Exercise 2 to validate that the properties to be used to send e-mail have the correct format.

|  |
| --- |
| Activity Validators An activity validator is a part of the Activity component model. The component model takes care of the following aspects of an activity:   * **Designer**: This component defines the visual representation of the activity on the workflow and custom activity designers. * **Code generator**: This is an extensibility point to generate custom code for an activity in a workflow. **InvokeWebService**, which is an out-of-the-box activity, has such an implementation. * **Validator**: This component enforces the activity semantics both at the design time and run time (only during dynamic update). * **Toolbox item**: Defines the custom behavior of the toolbox item representing the activity in a design environment like Visual Studio. * **Executor**: This is a stateless component which implements the execution semantics of an activity. This component may not be defined for simple cases, where the *Execute()* method of the activity is overridden as it was the case in the previous exercise. * **Serializer**: Provides customized serialization behavior if needed. * **Deployer**: This code is run when a workflow including the activity is deployed to a hosting environment. |

## Task 1 – Adding an Email Validation Code Activity

1. In the **Solution Explorer**, right click on *SendMailActivityLibrary* project to bring up the context menu.
2. Select the **Add | New Item…** menu item and add a “**Code File**” set the **Name** property to *ParametersValidator.cs*.

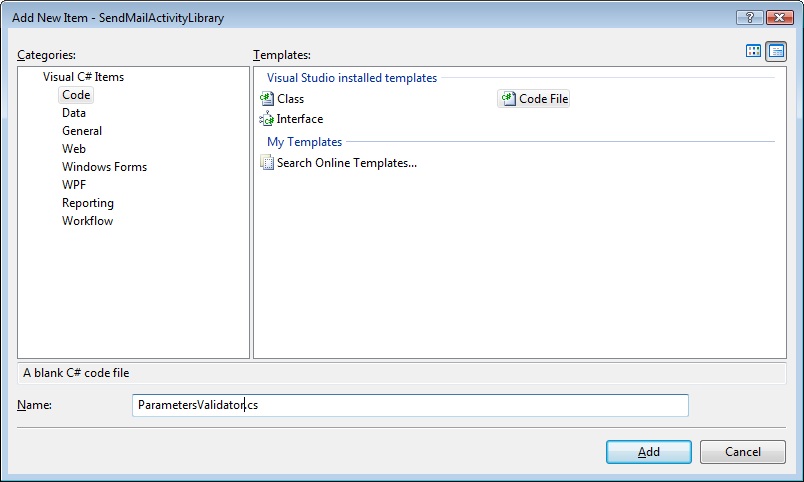


Figure : Add new Code File dialog

1. Replace the code in the new *ParametersValidator.cs*file with the following code snippet.

|  |
| --- |
| **Snippet:** WFLab02\_Ex03\_Task01\_ParametersValidator |
| **using System;**  **using System.Workflow.Activities;**  **using System.Workflow.ComponentModel;**  **using System.Workflow.ComponentModel.Compiler;**  **using System.Text.RegularExpressions;**  **using System.Net.Mail;**  **namespace SendMailActivityLibrary**  **{**  **public class ParametersValidator : ActivityValidator**  **{**  **}**  **}** |

|  |
| --- |
| About ActivityValidator Classes An **ActivityValidator** class contains both design-time and run-time validation logic to ensure that an activity is appropriately configured. User-defined validation classes inherit from either **ActivityValidator** or **CompositeActivityValidator** base classes, depending upon whether your activity is a basic or composite activity.  Validation occurs automatically during the compilation of a workflow and at runtime when a dynamic update of the activity is performed. |

1. Override the **ValidateProperties** method enabling the class to perform custom validation on the properties of **SendMailActivity**. Do this with the code snippet below.

The **ValidateProperties** method of an activity is used primarily for verifying compile-time semantics of the properties of an activity. This validation component will check that the e-mail addresses have the correct format. Enter the following into the *ParametersValidator* class.

|  |
| --- |
| **Snippet:** WFLab02\_Ex03\_Task01\_ValidateProperties |
| **public override ValidationErrorCollection**  **ValidateProperties(ValidationManager manager, object obj)**  **{**  **. . .**  **return validationErrors;**  **}** |

1. You may have noticed a call to a method called *IsValidEmailAddress.* This is the engine of the validation, it checks the strings it is given to verify whether it is a valid email address or not. Enter the following into the *ParametersValidator* class.  
   The *IsValidEmailAddress* method must be added to the *ParametersValidator* class. Add the following code snippet to the class.

|  |
| --- |
| **Snippet:** WFLab02\_Ex03\_Task01\_IsValidEmailAddress |
| **public Boolean IsValidEmailAddress(String address)**  **{**  **. . .**  **return true;**  **}** |

#### Using the new validator class in the activity

Using the new validator class is a matter of decorating the Activity class to be validated with the appropriate attribute, which will specify the validator to be invoked by the runtime.

1. Right click *SendMailActivity.*cs. Select **View Code**
2. Add the following **ActivityValidator** attribute to the *SendMailActivity*, as the yellow highlighted line of code shows:

|  |
| --- |
| **namespace SendMailActivityLibrary**  **{**  [ActivityValidator(typeof(ParametersValidator))]  **public partial class SendMailActivity :**  **System.Workflow.ComponentModel.Activity**  **{**  **. . .**  **}**  **}** |

1. Select the **Build | Build** **Solution** menu command.

## Task 2 – Testing The Validator

1. Double click on the file *SendMailWorkflow.cs* in the **Solution Explorer**.
2. Click on the **sendMail** activity.
3. Click on the **From** property for this activity. Enter an invalid email address.
4. Notice that a red circle with a contained exclamation point appears on the activity indicating the property validation error.

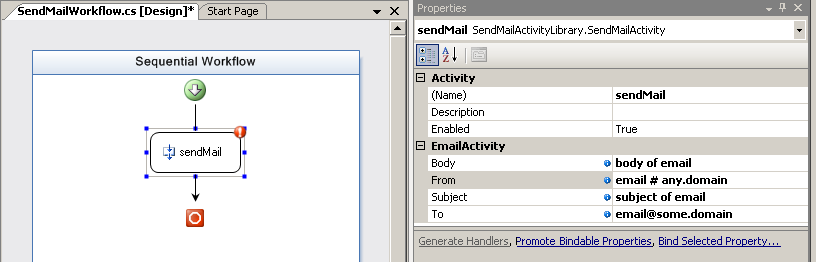


Figure : Design time validator in action.

1. Select the **Build | Build Solution** menu command.
2. Notice that you will receive a compile error telling you about the invalid email address.

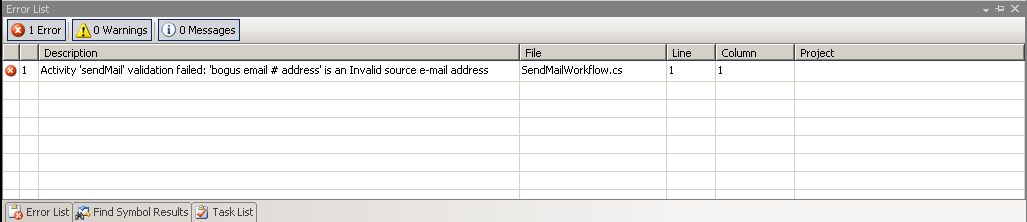


Figure : Compile time validation error

1. Set **From** to [*email@any.domain*](mailto:email@any.domain).

The design time validation passes now.

1. Select **File | Save All**.

# Exercise 4 – Adding an Activity Designer

Functionality can be added to custom activities to render a design time view.

This exercise enhances the *SendMailActivity* by creating a custom design-time interface for it. An **Activity Designer** defines the visual representation of the activity on the workflow and custom activity designers.

## Task 1 – Add the Activity Designer

#### Creating the activity designer class file

1. In the **Solution Explorer**, right click on *SendMailActivityLibrary* project. Select the **Add | Class** menu item.
2. Name the class file *SendMailDesigner.cs*.
3. Replace all code in the *SendMailDesigner.cs* file with the following code snippet.

This code sets up the *SendMailDesigner* class as a descendant of **ActivityDesigner**.

|  |
| --- |
| **Snippet:** WFLab02\_Ex04\_Task01\_SendMailDesigner |
| **using System;**  **using System.Text;**  **using System.ComponentModel;**  **using System.ComponentModel.Design;**  **using System.Collections;**  **using System.Drawing;**  **using System.Drawing.Drawing2D;**  **using System.Drawing.Design;**  **using System.Workflow.ComponentModel.Compiler;**  **using System.Workflow.ComponentModel.Serialization;**  **using System.Workflow.ComponentModel;**  **using System.Workflow.ComponentModel.Design;**  **using System.Workflow.Runtime;**  **using System.Workflow.Activities;**  **using System.Workflow.Activities.Rules;**  **namespace SendMailActivityLibrary**  **{**  **public class SendMailDesigner : ActivityDesigner**  **{**  **}**  **}** |

The class needs to override three methods, **Initalize*,* OnPaint**, and **OnLayoutSize**. The next steps detail what needs to happen and where.

#### Overriding Initialize

1. Override **Initalize** to save a reference to the **Activity** that we are going to perform custom drawing upon. The base class must also be allowed to initialize.

Enter the following snippet into the *SendMailDesigner* class.

|  |
| --- |
| **Snippet:** WFLab02\_Ex04\_Task01\_Initalize |
| **public class SendMailDesigner : ActivityDesigner**  **{**  SendMailActivity parentActivity;  protected override void Initialize(Activity activity)  {  base.Initialize(activity);  parentActivity = (SendMailActivity)activity;  }  **}** |

**Overriding OnLayoutSize**

1. Override **OnLayoutSize** to provide information regarding the size of the custom drawn object.

Enter the following snippet into the *SendMailDesigner* class.

|  |
| --- |
| **Snippet:** WFLab02\_Ex04\_Task01\_OnLayoutSize |
| **SendMailActivity parentActivity;**  **protected override void Initialize(Activity activity)**  **{ . . . }**  protected override Size OnLayoutSize(ActivityDesignerLayoutEventArgs e)  {  return new Size(230, 100);  } |

#### Overriding OnPaint

1. Override **OnPaint** in order to perform the custom drawing of the activity.

Enter the following snippet into the *SendMailDesigner* class

|  |
| --- |
| **Snippet:** WFLab02\_Ex04\_Task01\_OnPaint |
| **protected override Size OnLayoutSize(ActivityDesignerLayoutEventArgs e)**  **{**  **return new Size(230, 100);**  **}**  protected override void OnPaint(ActivityDesignerPaintEventArgs e)  {  Rectangle frameRect = new Rectangle(this.Location.X,  . . .  } |

#### Finishing the SendMailDesigner class

1. The utility method *RoundedRect* is used to round the corners and provide a sleeker interface. Paste the following into the *SendMailDesigner* class

Enter the following snippet into the *SendMailDesigner* class

|  |
| --- |
| **Snippet:** WFLab02\_Ex04\_Task01\_RoundedRect |
| **protected override void OnPaint(ActivityDesignerPaintEventArgs e)**  **{ . . . }**  private GraphicsPath RoundedRect(Rectangle frame)  {  . . .  return path;  } |

#### Specifying the designer class on the activity

1. Add the following “**Designer**” attribute to the *SendMailActivity*class defined in *SendMailActivity.cs*, as shown in the highlighted line of code.

|  |
| --- |
| **namespace SendMailActivityLibrary**  **{**  [Designer(typeof(SendMailDesigner), typeof(IDesigner))]  **[ActivityValidator(typeof(ParametersValidator))]**  **public partial class SendMailActivity :**  **System.Workflow.ComponentModel.Activity**  **{**  **. . .**  **}** |

1. Select the **Build | Build** **Solution** menu command to ensure the solution builds.

## Task 2 – Testing the Activity Designer

Now that design-time formatting functionality has been added to the custom activity, it can be seen in Visual Studio’s Visual Workflow Designer.

#### Testing the visual design experience

1. Double click on the file *SendMailWorkflow.cs* in the Visual Studio solution explorer.
2. The code for the customized drawing will execute and the *SendMailWorkflow* design surface should look like Figure 24.

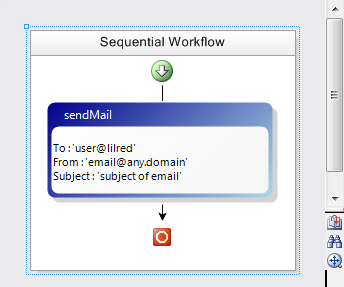


Figure : New design experience for custom activity

1. Click on the *SendMailActivity*activity.

#### Testing design-time property validation

1. Click on the **From** property for this activity.
2. Enter an invalid email address.
3. Notice that a red circle with a contained exclamation point appears on the activity indicating the property validation error, clicking on the smart tag will reveal details about the error.

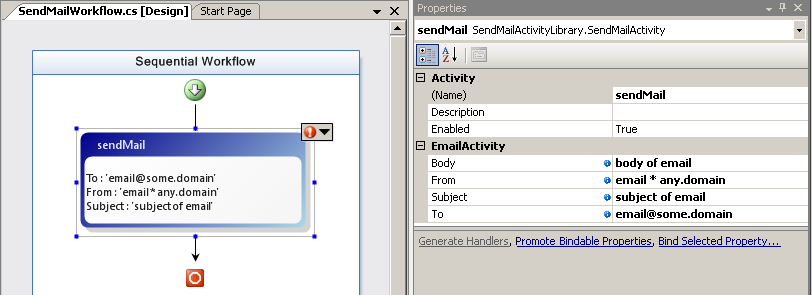


Figure : Invalid activity property at design time

1. Select the **Build | Build Solution** menu command.

Notice that you will receive a compile error telling you about the invalid email address.

1. Set From to [email@any.domain](mailto:email@any.domain).

The design time error disappears. The solution will now build.

1. Select the **File | Save All** menu command.
2. Select the **File | Close Solution** menu command.

#### Exercise Summary

This exercise used an **ActivityDesigner** to enhance the **SendMailActivity** to provide useful information on the activity for workflow developers.

# Lab Summary

This lab used Visual Studio 2008 to create a workflow that included a composite activity. This lab demonstrated the experience of creating a custom WF activity and adding functionality to make it behave intelligently at design-time. These techniques are very useful for creating professional and re-usable WF activities appropriate for re-distribution.

* You added custom properties to the activity to store values used to send e-mail.
* You created an executor method for the activity that used the custom properties to send e-mail.
* You added design-time validation to your activity to ensure the e-mail addresses were properly formatted.
* Finally you enhanced your *SendMailActivity* project by providing a customized user interface experience.