# Lab 11: Developing Custom Activities for a SharePoint Workflow Template

**Lab Overview:** Often there are pieces of code used many times in a solution. In a traditional .NET application, you encapsulate them as a class or a function. In Windows Workflow, you encapsulate them as Activities.

In this lab, you will be creating two custom activities that will provide database access. To do this you will need to create activities as well as validators and designers to ensure the activities are use properly.

## Exercise 1: Building a simple Activity

### Create a new **Workflow Activity Library** project to the solution named **SqlDatabaseActivities**.

#### Right click the solution in the **Solution Explorer** and click **Add -> New Project**.

#### Select the **Workflow** option in the **Project types** list box and select **Workflow Activity Library** in the Templates list.

#### Enter a name of **SqlDatabaseActivities** and click **OK**.

### Create a new **Activity** named **SqlScalarQueryActivity** in the new project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Activity**.

#### Enter a name of **SqlScalarQueryActivity.cs** and click **OK**.

#### Modify the class definition to change the base class to **Activity** instead of **SequenceActivity**.

public partial class SqlScalarQueryActivity : Activity

{

}

### Define the dependency properties named **Parameters** and **Result**.

#### Define the static **DependencyProperty** object for **Parameters**.

##### Create a new **public static** **DependencyProperty** object named **ParametersProperty**.

##### Set its value using the **DependencyProperty.Register** static method.

##### Define its **name** as **Parameters**, it’s **propertyType** as **typeof(SqlParameter[])**, and it’s **ownerType** of **typeof(SqlScalarQueryActivity)**.

public static DependencyProperty ParametersProperty =

DependencyProperty.Register("Parameters",

typeof(SqlParameter[]), typeof(SqlScalarQueryActivity));

#### Define a helper property that wraps access to the new dependency property.

##### Define a string **Parameters** property with a get and set seciton.

##### Implement get using the **base.GetValue** method and casting the result to a string.

##### Implement set using the **base.SetValue** method.

##### Add a **Category** attribute to the property with a value of **Parameters** (this places the property in the **Parameters** group in the designer).

[Category("Parameters")]

public SqlParameter[] Parameters

{

get { return base.GetValue(ParametersProperty) as SqlParameter[]; }

set { base.SetValue(ParametersProperty, value); }

}

#### Repeat the process to create a **Result** dependency property and wrapping property definition. The only difference is the type. Make **Result** an object.

public static DependencyProperty ResultProperty =

DependencyProperty.Register("Result",

typeof(System.Object), typeof(SqlScalarQueryActivity));

[Category("Parameters")]

public object Result

{

get { return base.GetValue(ResultProperty); }

set { base.SetValue(ResultProperty, value); }

}

### Define the **Query** dependency property. Make it a **MetaData** property so it can only be modified at design time.

#### Define the static **DependencyProperty** object for **Query**.

##### Create the new **public static** **DependencyProperty** just like before.

##### Add a fourth parameter to the call to **DependencyObject.Register** that is a **PropertyMetadata** object with an **options** parameter of **DependencyPropertyOptions.Metadata**.

public static DependencyProperty QueryProperty =

DependencyProperty.Register("Query",

typeof(System.String), typeof(SqlScalarQueryActivity),

new PropertyMetadata(DependencyPropertyOptions.Metadata));

#### Define the wrapper property for **Query** the same as any other dependency property wrapper property.

[Category("Parameters")]

public string Query

{

get { return base.GetValue(QueryProperty) as string; }

set { base.SetValue(QueryProperty, value); }

}

### Add two events to the activity, **Invoking** and **Invoked**, which can be bound using the same mechanism as the properties.

#### Define the static **DependencyProperty** object for **Invoking**.

##### Create a new **public static** **DependencyProperty** object named **InvokingEvent**.

##### Set its value using the **DependencyProperty.Register** static method.

##### Define its **name** as **Invoked**, it’s **propertyType** as **typeof(EventHandler)**, and it’s **ownerType** of **typeof(SqlScalarQueryActivity)**.

public static DependencyProperty InvokingEvent =

DependencyProperty.Register("Invoking",

typeof(EventHandler), typeof(SqlScalarQueryActivity));

#### Define the wrapper event for **Invoking** that will provide a standard mechanism for attaching and removing event handlers.

##### Define an event named **Invoked** of type **EventHandler** with an add and remove seciton.

##### Implement **add** using the **base.AddHandler** method.

##### Implement **remove** using the **base.RemoveHandler** method.

##### Add a **Category** attribute to the property with a value of **Handlers**.

[Category("Handlers")]

public event EventHandler Invoking

{

add { base.AddHandler(InvokingEvent, value); }

remove { base.RemoveHandler(InvokingEvent, value); }

}

#### Define a second dependency property and event for the **Invoked** event.

public static DependencyProperty InvokedEvent =

DependencyProperty.Register("Invoked",

typeof(EventHandler), typeof(SqlScalarQueryActivity));

[Category("Handlers")]

public event EventHandler Invoked

{

add { base.AddHandler(InvokedEvent, value); }

remove { base.RemoveHandler(InvokedEvent, value); }

}

### Implement the **Execute** method so the query is made and the result is stored in the **Result** property.

#### Override the **Execute** method of the **Activity** class.

protected override ActivityExecutionStatus Execute(ActivityExecutionContext executionContext)

{

}

#### Raise the **Invoking** event using the **Activity.RaiseEvent** method.

// raise the invoking event

this.RaiseEvent(InvokingEvent, this, EventArgs.Empty);

#### Open the **SqlConnection** using a hardcoded connection string (this will be changed later)

using (SqlConnection connection = new SqlConnection("Data Source=.\\SQLEXPRESS;AttachDbFilename=C:\\Labs\\Files\\Litware.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True"))

{

connection.Open();

}

#### While the connection is open, create and execute the new **SqlCommand** based on the **Query** and **Parameters** dependency properties.

// create the sql command

SqlCommand command = new SqlCommand(this.Query, connection);

if (this.Parameters != null)

foreach (SqlParameter parameter in this.Parameters)

command.Parameters.Add((parameter as ICloneable).Clone() as SqlParameter);

this.Result = command.ExecuteScalar();

#### After the **SqlConnection’s** using scope, raise the **Invoked** event to indicate the activity is completed.

// raise the Invoked event

this.RaiseEvent(InvokedEvent, this, EventArgs.Empty);

#### Return an **ActivityExecutionStatus** of **Closed** to let the executor know this activity is completed and it should move on to the next activity.

// return the closed activity status

return ActivityExecutionStatus.Closed;

### Build the **SqlDatabaseActivities** project.

#### Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

### Create a new **Sequential Workflow Console Application** project to the solution named **SqlDatabaseActivitiesConsole**.

#### Click **File -> New Project** in the menu bar.

#### Select the **Workflow** option in the **Project types** list box and select **Sequential Workflow Console Application** in the **Templates** list.

#### Select **Add to Solution** in the **Solution** drop down list.

#### Enter a name of **SqlDatabaseActivitiesConsole** and click **OK**.

### Add a new **SqlScalarQueryActivity** to **Workflow1** and use it to make a database query.

#### View **Workflow1.cs** in design mode by right clicking it in the **Solution Explorer** and selecting **View Designer**.

#### Drag a new **SqlScalarQueryActivity** onto the canvas and name it **LookupComments**.

##### Drag a **SqlScalarQueryActivity** onto the canvas.

##### In the properties pane, set the **Name** property to **LookupComments**.

#### Define the query as a lookup into the **UserProfiles** table.

##### In the properties pane, set the **Query** to the following query.

SELECT Comments FROM UserProfiles WHERE Email = @email

#### Define an **Invoking** method that will be used to define the **@email** parameter.

##### In the properties pane, set the **Invoking** property to **LookupComments\_Invoking** and press **Enter**.

##### In the **LookupComments\_Invoking** method, cast the sender parameter to a **SqlScalarActivityQuery**.

##### Set the activities **Parameters** property to an array containing a single **SqlParameter** named **@email** with a value of [**Administrator@litwareinc.com**](mailto:Administrator@litwareinc.com).

private void LookupComments\_Invoking(object sender, EventArgs e)

{

SqlScalarQueryActivity activity = sender as SqlScalarQueryActivity;

activity.Parameters = new SqlParameter[]

{

new SqlParameter("@email", "Administrator@litwareinc.com")

};

}

#### Define an **Invoked** event handler that will be used to display the activity’s Result property to the console.

##### Back in the designer’s properties pane, set the **Invoked** property to **LookupComments\_Invoked** and press **Enter**.

##### In the **LookupComments\_Invoked** method, cast the sender parameter to a **SqlScalarActivityQuery**.

##### Use **Console.WriteLine** to display the activity’s **Result** property to the console.

private void LookupComments\_Invoked(object sender, EventArgs e)

{

SqlScalarQueryActivity activity = sender as SqlScalarQueryActivity;

Console.WriteLine("Result - {0}", activity.Result);

}

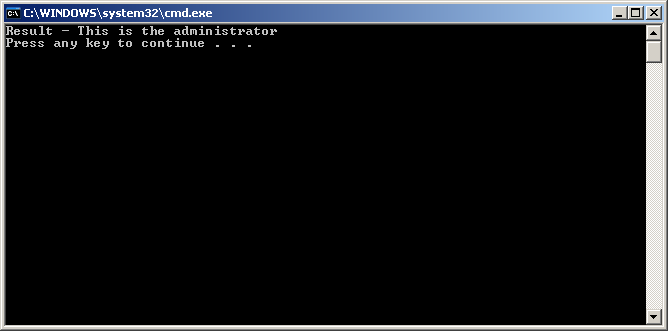
### Run the workflow and verify that the output displays the appropriate result.

#### Verify the **SqlDatabaseActivitiesConsole** is bold in the **Solution Explorer**.

##### If not, right click it and select **Set as Startup Project**.

#### Build and execute the workflow.

##### Click **Debug -> Start Without Debugging**.



## Exercise 2: Creating a Composite Activity

### Create a new **Activity** named **SqlConnectionActivity** in the **SqlDatabaseActivities** project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Activity**.

#### Enter a name of **SqlConnectionActivity.cs** and click **OK**.

### Define the **SqlConection** property that will be used to expose the connection to other components.

#### Define a property with a public getter and private setter.

public SqlConnection Connection { get; private set; }

#### Add a **Browsable** attribute to the property so it does not show up in the designer.

[Browsable(false)]

public SqlConnection Connection { get; private set; }

### Define the dependency property named **ConnectionString** of type **string**.

public static DependencyProperty ConnectionStringProperty =

DependencyProperty.Register("ConnectionString",

typeof(System.String), typeof(SqlConnectionActivity));

[Category("Parameters")]

public string ConnectionString

{

get { return base.GetValue(ConnectionStringProperty) as string; }

set { base.SetValue(ConnectionStringProperty, value); }

}

### Implement the execute method to start the processing of the child activities.

#### Override the **Execute** method of the **Activity** class.

protected override ActivityExecutionStatus Execute(ActivityExecutionContext executionContext)

{

}

#### If no child activities exist that are enabled, return with an **ActivityExecutionStatus** of **Closed**.

// check if there's any activities to execute

if (base.EnabledActivities.Count == 0)

return ActivityExecutionStatus.Closed;

#### Create and open a new **SqlConnection** using the **ConnectionString** dependency property.

// open the database connection

if (this.Connection == null)

{

this.Connection = new SqlConnection(this.ConnectionString);

this.Connection.Open();

}

#### Find the first activity in the list of enabled activities, start it and register for status change message from that activity.

##### Access and store the first item in the **base.EnabledActivities** collection

##### Use the **Activity.RegisterForStatusChange** method to register for the **ClosedEvent**.

##### Use the **ActivityExecutionContext.ExecuteActivity** to start the first activity executing.

// start the first child activity and register for it's state change event

Activity firstEnabledActivity = base.EnabledActivities[0];

firstEnabledActivity.RegisterForStatusChange(Activity.ClosedEvent, this);

executionContext.ExecuteActivity(firstEnabledActivity);

#### Return an **ActivityExecutionStatus** of **Executing** to notify the runtime that this activity is still running.

// tell the runtime this activity is still executing

return ActivityExecutionStatus.Executing;

### Implement the **IActivityEventListener<ActivityExecutionStatusChangedEventArgs>** interface so the activity can receive activity events from its child activities.

#### Modify the class to derive from **IActivityEventListener<ActivityExecutionStatusChangedEventArgs>**.

public partial class SqlConnectionActivity: SequenceActivity,

IActivityEventListener<ActivityExecutionStatusChangedEventArgs>

#### Implement the **OnEvent** method to handle the receipt of a status change event from a child activity.

public void OnEvent(object sender, ActivityExecutionStatusChangedEventArgs e)

{ }

#### Validate that the sender parameter of the **OnEvent** method is an **ActivityExecutionContext**.

// validate the sender

ActivityExecutionContext executionContext = sender as ActivityExecutionContext;

if (executionContext == null)

throw new ArgumentException("sender");

#### Using the **Activity.UnregisterForStatusChange** method, cancel the subscription to events from the activity that has closed.

// unregister the handler for status events

e.Activity.UnregisterForStatusChange(Activity.ClosedEvent, this);

#### If the current activity’s **ExecutionStatus** property indicates a **Canceling** or **Faulting** state, close the activity.

// if the activity is cancelling or faulting, cleanup

if ((this.ExecutionStatus == ActivityExecutionStatus.Canceling) || (this.ExecutionStatus == ActivityExecutionStatus.Faulting)) {

executionContext.CloseActivity();

}

#### If the current activity’s **ExecutionStatus** property indicates that the activity is in the **Executing** state, find the next activity and execute it.

##### Use the **FindNextChild** method to lookup the next child activity to be executed (this will be implemented in the next step.

##### If another activity was found, register to the closed event and tell the executor to execute the activity.

##### If no more activities were found, tell the execution context to close the **SqlConnectionActivity**.

// if the activity is executing, schedule the next child

else if (this.ExecutionStatus == ActivityExecutionStatus.Executing) {

// find the next activity in the sequence

Activity nextActivity = FindNextChild();

if (nextActivity != null) {

// register and start the next activity

nextActivity.RegisterForStatusChange(Activity.ClosedEvent, this);

executionContext.ExecuteActivity(nextActivity);

}

else {

// close this activity

executionContext.CloseActivity();

}

}

### Create a **FindNextActivity** method that will find the next child activity to execute.

#### Define a private method named **FindNextChild** that returns an **Activity** object.

private Activity FindNextChild()

{

}

#### Check if there are any enabled activities in the list, if not return null.

// check if any activities are in the list

if (base.EnabledActivities.Count == 0)

return null;

#### Find the last activity in the list that has a status of closed.

// find the last activity in the list that is closed

Activity lastClosedActivity = base.EnabledActivities.LastOrDefault(

n => n.ExecutionStatus == ActivityExecutionStatus.Closed);

#### If the last closed activity is the last activity in the list, return **null**.

// if this is the last closed activity, return false

int indexOfLastClosedActivity =

base.EnabledActivities.IndexOf(lastClosedActivity);

if (indexOfLastClosedActivity == base.EnabledActivities.Count - 1)

return null;

#### Return the activity following the last closed activity.

// return try to indicate a task was found

return base.EnabledActivities[indexOfLastClosedActivity + 1];

### Handle the activity’s **OnClosed** method to make sure the **SqlConnection** is closed.

#### Override the **OnClosed** method of the **Activity** class.

protected override void OnClosed(IServiceProvider provider)

{

}

#### If the connection is not null, close it (if necessary) and set it to null.

// close the database connection

if (this.Connection != null)

{

if (this.Connection.State != System.Data.ConnectionState.Closed)

this.Connection.Close();

this.Connection = null;

}

#### Call the base class’s implementation.

// call the base implementation

base.OnClosed(provider);

### Update the **SqlScalarQueryActivity** to use the parent **SqlConnectionActivity’s** **SqlConnection** instead of creating its own.

#### Open the code for **SqlScalarQueryActivity** by right clicking it in the **Solution Explorer** and selecting **View Code**.

#### Add a private method named **FindParentConnection** that accepts searches up the activity tree and finds the first **SqlConnectionActivity**.

##### Use the **Activity.Parent** property to find the parent.

##### If the parent is a **SqlConnectionActivity**, return it.

##### Set the **activity** variable to its parent, and keep looking.

private SqlConnectionActivity FindParentConnection(Activity current)

{

while (current.Parent != null)

{

if (current.Parent is SqlConnectionActivity)

return current.Parent as SqlConnectionActivity;

current = current.Parent;

}

return null;

}

#### In the **Execute** method, remove all code that creates or opens the **SqlConnection** object.

#### Immediately following the call where the **InvokingEvent** is raised, add the code to find the parent **SqlConnectionActivity** using the **FindParentConnection**.

// find the parent SqlConnectionActivity

SqlConnectionActivity connectionActivity = FindParentConnection(this);

if (connectionActivity == null)

throw new InvalidOperationException("No parent connection was found.");

#### Using the parent **SqlConnectionActivity’s** Connection property, store the **SqlConnection** in a variable named connection.

// store the connection for later

SqlConnection connection = connectionActivity.Connection;

### Build the **SqlDatabaseActivities** project.

#### Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

### Add a new **SqlConnectionActivity** to **Workflow1** and move the **SqlScalarQueryAction** into it.

#### View **Workflow1.cs** in design mode by right clicking it in the **Solution Explorer** and selecting **View Designer**.

#### Drag a new **SqlConnectionActivity** onto the canvas and name it **SqlConnection**.

##### Drag a **SqlConnectionActivity** onto the canvas.

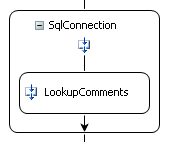
##### In the properties pane, set the **Name** property to **SqlConnection**.

#### Define the connection string property as a lookup into the **UserProfiles** table.

##### In the properties pane, set the **ConnectionString** propertyto the following value.

Data Source=.\SQLEXPRESS;AttachDbFilename=C:\Labs\Files\Litware.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True

#### Drag the existing **SqlScalarQueryActivity** into the new **SqlConnection** activity.



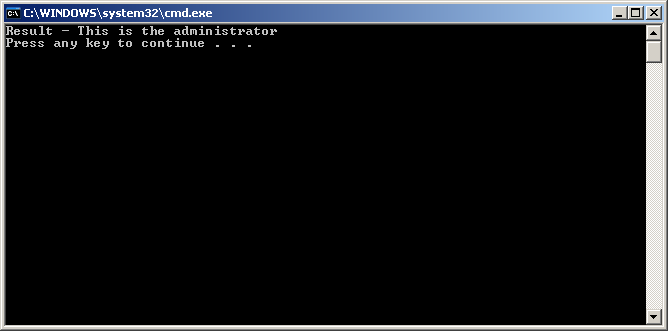
### Run the workflow and verify that the output displays the appropriate result.

#### Verify the **SqlDatabaseActivitiesConsole** is bold in the **Solution Explorer**.

##### If not, right click it and select **Set as Startup Project**.

#### Build and execute the workflow. The result should be the same as before, but now the connection is shared.

##### Click **Debug -> Start Without Debugging**.



## Exercise 3: Adding Validation to the Activities

### Create a new **class** named **SqlScalarQueryActivityValidator** in the new project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.

#### Enter a name of **SqlScalarQueryActivityValidator.cs** and click **OK**.

#### Modify the class definition so it is **public** and derives from **ActivityValidator**.

public class SqlScalarQueryActivityValidator : ActivityValidator

{

}

#### Open the code for the **SqlScalarQueryActivity** and attach the **Validator** attribute defining the activity’s validator.

##### Open the code for **SqlScalarQueryActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.

##### Attach the **Validator** attribute to the **SqlScalarQueryActivity** class using the type of the new. **SqlScalarQueryActivityValidator** class as the parameter.

[ActivityValidator(typeof(SqlScalarQueryActivityValidator))]

public partial class SqlScalarQueryActivity : Activity

### Validate that the **Query** property is not **null** or an empty string.

#### Override the **Validate** method from the **ActivityValidator** class.

public override ValidationErrorCollection Validate(ValidationManager manager, object obj)

{ }

#### Cast the **obj** parameter to a **SqlScalarQueryActivity** object and store it for later.

SqlScalarQueryActivity activity = obj as SqlScalarQueryActivity;

#### Run the base validator and store any errors in a new errors variable of type **ValidationErrorCollection**.

// build the errors collection using the base validation method

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

#### If the validator is running in a real environment (not as part of the activity’s designer), validate that the **Query** property is not null or empty.

##### Check if the **activity.Parent** attribute is not null. If it isn’t, continue the validation.

##### Check if the **activity.Query** property is null or empty using **string.IsNullOrEmpty** method.

##### If the query property is empty, create a new validation error using the **ValidationError.GetNotSetValidationError** method and add it to the **error** collection.

// only perform validation if in design or runtime mode

if (activity.Parent != null) {

// validate that a query was defined

if (string.IsNullOrEmpty(activity.Query))

errors.Add(ValidationError.GetNotSetValidationError("Query"));

}

#### Return the collection of errors.

// return the errors

return errors;

### Verify the **SqlScalarQueryActivity** has a parent **SqlConnectionActivity**.

#### Create a new method named **HasSqlConnectionParent** that looks through an activity’s parents to find an object of a specific type.

##### Use the **Activity.Parent** property to find the parent.

##### If the parent is a **SqlConnectionActivity**, return **true**.

##### Set the activity variable to its parent, and keep looking.

private bool HasSqlConnectionParent(Activity current) {

while (current.Parent != null) {

if (current.Parent is SqlConnectionActivity)

return true;

current = current.Parent;

}

return false;

}

#### In the **Validate** method, immediately after validating the **Query** parameter, check if the activity has a **SqlConnectionActivity** as a parent.

// validate the activity has an ancestor that is a SqlConnectionActivity

if (!HasSqlConnectionParent(activity))

errors.Add(new ValidationError(

"SqlScalarQueryActivities must be palaced in a SqlConnectionActivity",

100, false));

### Create a new class named **SqlConnectionActivityValidator** in the **SqlDatabaseActivities** project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.

#### Enter a name of **SqlConnectionActivityValidator.cs** and click **OK**.

#### Modify the class definition so it derives from **ActivityValidator**.

public class SqlConnectionActivityValidator : CompositeActivityValidator

{ }

#### Open the code for the **SqlConnectionActivity** and attach the **Validator** attribute.

##### Open the code for **SqlConnectionActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.

##### Attach the **Validator** attribute to the **SqlConnectionActivity** class using the type of the new **SqlConnectionActivityValidator** class as the parameter.

[ActivityValidator(typeof(SqlConnectionActivityValidator))]

public partial class SqlConnectionActivity : CompositeActivity

### Validate that the **ConnectionString** property is not null or an empty string.

#### Override the **Validate** method from the **ActivityValidator** class.

public override ValidationErrorCollection Validate(ValidationManager manager, object obj)

{

}

#### Cast the **obj** parameter to a **SqlScalarQueryActivity** object and store it for later.

SqlConnectionActivity activity = obj as SqlConnectionActivity;

#### Run the base validator and store any errors in a new errors variable of type **ValidationErrorCollection**.

// build the errors collection using the base validation method

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

#### If the validator is running in a real environment (not as part of the activity’s designer), validate that the **ConnectionString** property is not null or empty.

##### Check if the **activity.Parent** attribute is not null. If it isn’t, continue the validation.

##### Check if the **activity.ConnectionString** property is null or empty using **string.IsNullOrEmpty** method.

##### If the connection stringproperty is empty, create a new validation error using the **ValidationError.GetNotSetValidationError** method and add it to the **error** collection.

// only perform validation if in design or runtime mode

if (activity.Parent != null)

{

// validate that a connection string was defined

if (string.IsNullOrEmpty(activity.ConnectionString))

errors.Add(ValidationError.GetNotSetValidationError("ConnectionString"));

}

#### Return the collection of errors.

// return the errors

return errors;

### Verify the **SqlConnectionActivity** does not have any child activities that derive from **IEventHandler**.

#### Create a new method named **HasChildIEventActivity** that looks through a composite activity’s child activities and checks if any derive from **IEventHandler**.

private bool HasChildIEventActivity(CompositeActivity activity)

{

}

#### Loop through each enabled child activity and check if the activity derives from **IEventHandler**. If so, return **true**.

#### Check each child activity to determine if it is a **CompositeActivity**. If so, call **HasChildIEventActivity** and return true if it returns **true**.

// loop through each child activity that is enabled

foreach (Activity childActivity in activity.EnabledActivities) {

// if the child activity is an IEventActivity, log an error

if (childActivity is IEventActivity)

return true;

// if the child activity is a composite activity, check its children

CompositeActivity compositeActivity = childActivity as CompositeActivity;

if (compositeActivity != null)

if (HasChildIEventActivity(compositeActivity))

return true;

}

#### I no child activities are **IEventActivities**, return **false**.

// no child activities are IEventActivity

return false;

#### In the **Validate** method, immediately after validating the **ConnectionString** parameter, check if the activity has a **SqlConnectionActivity** as a parent.

// validate the activity has has no child IEventActivity activities

if (HasChildIEventActivity(activity))

errors.Add(new ValidationError(

"SqlConnectionActivities cannot contain IEventActivity activities. ",

100, false));

### Build the **SqlDatabaseActivities** project.

#### Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

### Test the validators by trying to create activities that are missing key properties.

#### Open **Workfow1.cs** in design mode by right clicking it and clicking **View Designer**.

#### Drag a new **SqlConnectionActivity** onto the canvas.

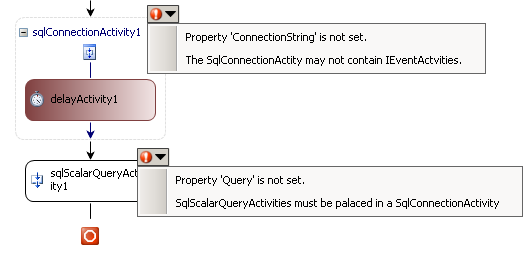
#### Drag a new **Delay** activity into the new **SqlConnectionActivity**.

#### Drag a new **SqlScalarQueryActivity** onto the canvas.

#### Verify each of the validators has fired.

##### The **SqlConnectionActivity** should have an error indicating an empty **ConnectionString** and containing an **IEventActiity**.

##### The **SqlScalarQueryActivity** should have an error indicating an empty query and no parent **SqlConnectionActivity**.



#### Clean up by deleting the activities just added.

## Exercise 4: Adding custom designers to the Activities

### Create a new classnamed **SqlScalarQueryActivityDesigner** in the **SqlDatabaseActivities** project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.

#### Enter a name of **SqlScalarQueryActivityDesigner.cs** and click **OK**.

#### Modify the class definition so it is **public** and derives from **ActivityDesigner**.

public class SqlScalarQueryActivityDesigner : ActivityDesigner

{

}

#### Open the code for the **SqlScalarQueryActivity** and attach the **Designer** attribute defining the activity’s designer.

##### Open the code for **SqlScalarQueryActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.

##### Attach the **Designer** attribute to the **SqlScalarQueryActivity** class using the type of the new **SqlScalarQueryActivityDesigner** class as the parameter.

[Designer(typeof(SqlScalarQueryActivityDesigner))]

public partial class SqlScalarQueryActivity : Activity

### Restrict **SqlScalarQueryActivities** from being placed on the canvas unless it is in a **SqlConnectionActivity**.

#### Create a new method named **HasSqlConnectionParent** that looks through an activity’s parents to find an object of a specific type.

##### Use the **Activity.Parent** property to find the parent.

##### If the parent is a **SqlConnectionActivity**, return **true**.

##### Set the **activity** variable to its parent, and keep looking.

private bool HasSqlConnectionParent(Activity current)

{

while (current.Parent != null)

{

if (current.Parent is SqlConnectionActivity)

return true;

current = current.Parent;

}

return false;

}

#### Override the **CanBeParentedTo** method to restrict the activities that this activity can be contained in.

##### Define the **CanBeParentedTo** method.

##### If the **Activity** property in the parent designer object is of type **SqlConnectionActivity**, return **true**.

##### If not, check if any of the parent objects are of type **SqlConnectionActivity**.

public override bool CanBeParentedTo(CompositeActivityDesigner parentActivityDesigner)

{

// if the parent activity is a SqlConnectionActivity

if (parentActivityDesigner.Activity is SqlConnectionActivity)

return true;

// if not, check if a parent is a SqlConnectionActivity

return HasSqlConnectionParent(parentActivityDesigner.Activity);

}

### Create a new classnamed **SqlConnectionActivityDesigner** in the **SqlDatabaseActivities** project.

#### Right click on the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Add -> Class**.

#### Enter a name of **SqlConnectionActivityDesigner.cs** and click **OK**.

#### Modify the class definition so it is **public** and derives from **SequentialActivityDesigner**.

public class SqlConnectionActivityDesigner : SequentialActivityDesigner

{

}

#### Open the code for the **SqlDesignerActivity** and attach the **Designer** attribute defining the activity’s designer.

##### Open the code for **SqlConnectionActivity.cs** by right clicking it in the **Solution Explorer** and clicking **View Code**.

##### Attach the **Designer** attribute to the **SqlConnectionActivity** class using the type of the new **SqlConnectionActivityDesigner** class as the parameter.

[Designer(typeof(SqlConnectionActivityDesigner))]

public partial class SqlConnectionActivity : SequenceActivity

### Restrict the activities that can be placed in the **SqlConnectionActivity**.

#### Override the **CanInsertActivities** method in the **SqlConnectionActivityDesigner**.

#### If any of the activities in **activitiesToInsert** parameter derive from **IEventActivity**, return **false**.

public override bool CanInsertActivities(HitTestInfo insertLocation, ReadOnlyCollection<Activity> activitiesToInsert)

{

// check if any activities are of type IEventActivity

return !activitiesToInsert.Any(n => n is IEventActivity);

}

### Add a new verb to the designer that will allow workflow developers to right click and add a new **SqlScalarQueryActivity** to the **SqlConnectionActivity**.

#### Create a private event handler named **AddScalarQuery\_Click**.

private void AddScalarQuery\_Click(object sender, EventArgs e) { }

#### Cast the designer’s activity to a **CompositeActivity** variable for later.

//Access the designer’s activity using the base classes Activity property.

CompositeActivity activity = base.Activity as CompositeActivity;

#### Create a list of new Activity objects and create a new **SqlScalarQueryActivity** in the list.

// create the list of new activites

List<Activity> activities = new List<Activity>();

activities.Add(new SqlScalarQueryActivity());

#### Create a **ConnectorHitTestInfo** object identifying where to add the activity. In this case add it as the last activity in the **SqlConnectionActivity**.

// determine where to add the activities

ConnectorHitTestInfo location =

new ConnectorHitTestInfo(this,

HitTestLocations.Designer, activity.Activities.Count);

#### Add the child activities using the base class’s **InsertActivities** method.

// add the activities to the designer

base.InsertActivities(location, activities.AsReadOnly());

#### Override the **Verbs** property.

protected override ActivityDesignerVerbCollection Verbs { get {} }

#### Create a new **ActivityDesignerVerbCollection** and populate it by calling the base class’s implementation.

// load the verbs from the base class

ActivityDesignerVerbCollection verbs = new ActivityDesignerVerbCollection();

verbs.AddRange(base.Verbs);

#### Add the new verb named **Add Scalar Query** the calls the **AddScalarQuery\_Click**.

// add the new verb

verbs.Add(

new ActivityDesignerVerb(this,DesignerVerbGroup.Actions, "Add Scalar Query",

new EventHandler(AddScalarQuery\_Click)));

#### Return the new list of verbs.

// return the list of verbs

return verbs;

### Build the **SqlDatabaseActivities** project.

#### Right click the **SqlDatabaseActivities** project in the **Solution Explorer** and click **Build**.

### Test the designers by trying to place activities in positions that are not valid

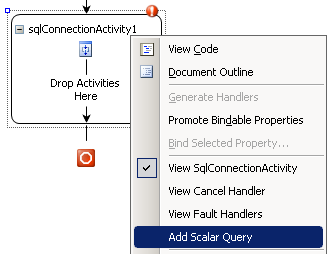
#### Open **Workfow1.cs** in design mode by right clicking it and clicking **View Designer**.

#### Drag a new **SqlConnectionActivity** onto the canvas.

#### Try to drag a new **Delay** activity into the new **SqlConnectionActivity** and verify it will not work.

#### Try to drag a new **SqlScalarQueryActivity** outside of the **SqlConnectionActivity** and verify it will not work.

#### Right click on the **SqlConnectionActivity** and select **Add Scalar Query** to add a new **SqlScalarQueryActivity** to the **SqlConnectionActivity**.



## Challenge: Designer Themes and Toolbox Items

Add a designer theme to the **SqlConnectionActivityDesigner** that changes the background color of the **SqlConnectionActivity**. To do this you will need to create a class derived from **CompositeDesignerTheme** and set the theme’s properties in the constructor. Once the theme is complete, attach it to the designer using the **ActivityDesignerTheme** attribute.

Add a toolbox item to the **SqlConnectionActivity** that adds a **SqlConnectionActivity** and an embedded **SqlScalarQueryActivity** when a **SqlConnectionActivity** is added to the canvas. To do this you will need to create a class derived from **ActivityToolboxItem** and override **CreateComponentsCore**. Once the toolbox item is complete, attach it to the activity using the **ToolboxItem** attribute.