be greater than zero. If it is defined dynamically through activity databinding, we cannot validate the value because it will be only determined at runtime.

Step 4

Here I will show you how to build a simple composite activity. To keep the focus on how custom activities behave, I will use code only, and not the designer. In this step, the Traffic Light activity is unchanged. The **Controller** composite activity class, which inherits from **CompositeActivity**, is called by the *CreateWorkflow* method of the **WorkflowRuntime** class to start the workflow. This activity models the controlling of a traffic intersection with two lights. One is red, while the other is green. The activity has a public property, *Iterations*, which controls the number of times the controller loops through the light cycle. Its value is set by the workflow host.

There are three methods of interest in the Controller activity: the constructor, the *Execute* method, and the delegate *ChildClosed*.

The constructor creates and initializes the values for the two traffic lights under control - the intersection of Main and Fifth. The two traffic lights have different timings. More traffic is on Fifth than Main so that light remains green longer. In addition, the two activities are added to the collection of activities associated with the Controller composite activity.

```
public Controller()
{
   Name = "Controller";

   mainStreet = new TrafficLightActivity();
   fifthAvenue = new TrafficLightActivity();
```

```
    mainStreet.Name = "MainStreet";
    mainStreet.Time = 1200;

    fifthAvenue.Name = "FifthAvenue";
    fifthAvenue.Time = 2000;

    CanModifyActivities = true;
    Activities.Add(mainStreet);
    Activities.Add(fifthAvenue);
    CanModifyActivities = false;
}
```

As before, the activity execution occurs in the Execute method. Each traffic light in the *EnabledActivites* collection is executed once for each iteration. The *EnabledActivites* collection is used instead of the *Activities* collection to which we added the activities in the constructor. As will be explained later, activites such as fault handlers are in the *Activities* collection, but not the *EnabledActivites* collection because they are not executed in the normal activity flow.

```
protected override ActivityExecutionStatus
                                Execute(ActivityExecutionContext context)
{
    if (EnabledActivities.Count == 0)
        return ActivityExecutionStatus.Closed;

    maximum = iterations * EnabledActivities.Count;

    for (int i = 0; i < iterations; i++)
    {
```

```
            foreach (TrafficLightActivity child in EnabledActivities)
            {
                ActivityExecutionContextManager manager =
                                              context.ExecutionContextManager;
                ActivityExecutionContext childContext =
                                         manager.CreateExecutionContext(child);
                childContext.Activity.Closed += new EventHandler<
                      ActivityExecutionStatusChangedEventArgs>(ChildClosed);
                childContext.ExecuteActivity(childContext.Activity);
            }
        }
        return ActivityExecutionStatus.Executing;
    }
```

An **ActivityExecutionContext** (AEC) class instance is passed into the Execute method. The AEC instance can be used to schedule activities and access workflow services. It represents the state of the workflow, and the tree of workflow activities at the moment it is created. This context is analogous to the concept of scope in a programming language. The Execute method of the activity at the root of the workflow (in this case the Controller activity) gets a context instance that is create by the workflow runtime. The lifecycle of this *default execution contex*t corresponds to the lifecycle of the workflow because its root is the initial activity executed when the *CreateWorkflow* method is invoked.

In looping situations, such as the one here, each iteration has to be independent of each other. Think of a dependency property that is bound to a value that can change. You must use the value for the current iteration. In such cases, as we saw with the while activity in the previous step, the

activity that controls the iteration must create an execution context for each iteration. The **ExecutionContextManager** class allows you to do this.

Using the execution context passed into the Execute method, an instance of the **ExecutionContextManager** class is created. This class is used to create an AEC instance for the child activity that is about to be executed. The controller activity then subscribes to the closed event for each of its child activities, and then schedules them for execution. Notice that they are all scheduled before any one of them executes (as discussed in the Part 3 shortcut).

Why subscribe to the close event? The controller cannot enter into the closed state until all its child activities have completed. So it must subscribe to these events and close when the last child close event occurs. Hence the *Execute* method returns *ActivityExecutionStatus.Executing* instead of *ActivityExecutionStatus.Closed*.

The *ChildClosed* delegate is called when the a Traffic Light activity closes. In this method, the child execution context is completed. When all the child Traffic Light activities have been closed the controller activity can close itself. Since the *ChildClosed* delegate has no return value, the *CloseActivity* method is used by the controller activity to enter the closed state.

```
void  ChildClosed(object sender,
                            ActivityExecutionStatusChangedEventArgs e)
{
   e.Activity.Closed -= ChildClosed;

   ActivityExecutionContext context =
                                    sender as ActivityExecutionContext;
   ActivityExecutionContextManager manager =
                                       context.ExecutionContextManager;
```

```
    ActivityExecutionContext childContext =
                                    manager.GetExecutionContext(e.Activity);
    manager.CompleteExecutionContext(childContext, false);

    count++;
    if (count == maximum)
        context.CloseActivity();
}
```

Figure 81 shows the results of running Step 4.

FIGURE 81
Running Step 4

Step 5

I have already discussed in the previous shortcut the scheduling of fault handlers. Now I will examine the details of how activities handle faults.