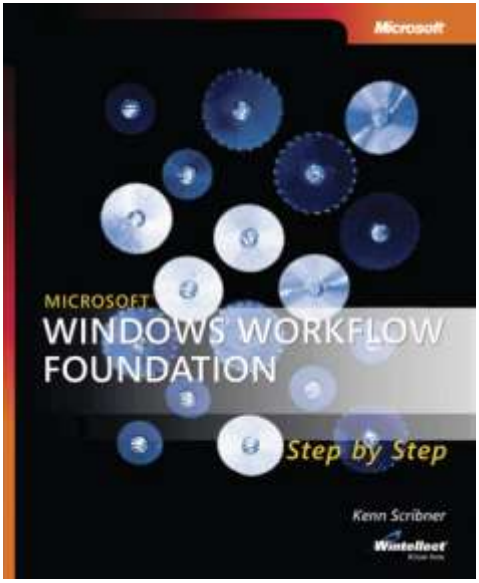


Microsoft® Windows® Workflow Foundation Step by Step

Kenn Scribner (Wintellect)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10023.aspx

9780735623354
Publication Date: February 2007

Microsoft Press

Table of Contents

Part II Working with Activities

Part III Workflow Processing

Chapter 14

State-Based Workflows

After completing this chapter, you will be able to:

- Understand the notional concept of a state machine and how it is modeled in workflow processing
- Create state-based workflows
- Apply initial and terminal state conditions
- Incorporate code to transition from state to state

In Chapter 4, "Introduction to Activities and Workflow Types," where I described the types of workflows you can create using Windows Workflow Foundation (WF), I mentioned the state-based workflow. State-based workflows model something known as the *finite state machine*. State-based workflows shine when the workflow requires much interaction with outside events. As events fire and are handled by the workflow, the workflow can transition from state to state as required.

WF provides a rich development experience for creating state-based workflows, and much of what you've seen in the book so far applies to state-based workflows. For example, when a state is transitioned into, you can, if you want, execute a few sequential activities, make conditional decisions (using rules or code), or iterate through some data points using an iterative activity structure. The only real difference is how the activities are queued for execution. In a sequential or parallel workflow, they're queued as they come up. But in a state-based workflow, activities are queued as states are transitioned into and out of. Events generally drive those transitions, but this is not a universal rule. Let's take another look at the conceptual state machine and relate those concepts to WF activities you can use to model your workflows.

The State Machine Concept

State machines are meant to model discrete points within your processing logic, the transitions to which are controlled by events. For example, you load your washing machine, close the door, and push the start button. Pushing the start button initiates a state machine that runs your laundry through the various cleaning cycles until the cycles are complete.

State machines have known starting points and known termination or end points. The states in between should be controlled by events expected to occur while the machine is at a specific state. Sometimes events throw state machines into invalid states, which are conditions not unlike sustaining unhandled exceptions in your applications. The entire process either comes

to a sudden halt or crashes entirely. Either way, transitioning to invalid states is something you want to monitor closely, at least in digital electronic systems. (WF is a bit more forgiving because you control when states are transitioned to by using an activity designed for the task—no transitional activity, no transition.)

Chapter 4 covered the essential concepts involved with state machines in general. For a quick refresher, see the section "The *State* Activity." Let's instead find out how activities designed to be used within state-based workflows are used.

Using the *State* Activity

Perhaps not too surprisingly, the *State* activity models a state in your state-based workflow. It's a composite activity, but it's limited to accepting only certain types of activities as children: the *EventDriven* activity, the *StateInitialization* activity, the *StateFinalization* activity, and other *State* activities. The *EventDriven* activity waits for the events that will cause a transition to another state, while *StateInitialization* and *StateFinalization* are activities guaranteed to execute when a state is transitioned into and out of, respectively. It might seem odd to be able to drop a secondary *State* activity into an existing *State* activity, but the intent is to provide the capability for embedding child state machines within parent state machines.

There is also a restriction regarding the number of valid activities your state can contain. Only a single instance of *StateInitialization* and *StateFinalization* is allowed. You can have one of each, but not more than one of each. Neither is required.

However, nothing says you can't have one or more instances of child *EventDriven* and *State* activities. In fact, it's common to find multiple *EventDriven* activities, because each event might cause a transition to a different state. For example, a "disapprove" event might transition to the final state, while an "approve" event might transition to a state designed to request more approval. As for *State* activities, clearly more than one should be allowed if you are to create embedded state-based workflows. *State*-based workflows with a single state model a simple sequential workflow, so in that case you should probably use a sequential workflow directly.

In any case, to use the *State* activity, simply drag an instance from the Toolbox onto the visual workflow designer. The only requirement is the workflow itself must be a state-based workflow rather than sequential. Then decide what child activities your state should maintain, and drag and drop them as required, keeping in mind the four types of activities you can insert.

Using the *SetState* Activity

In a purely electronic system, one made using electrical components (your computer's processor, for example), the fact that an event fires is enough to transition from one state to another. The presence of an electrical voltage or voltages sets in motion everything that's required for a state change.

WF is not an electronic system, even if it is executed on your system's processor. What this means to you is that firing an event your design specifies as the signal for a state change is *not* enough to transition to the other state. Instead, you must insert another activity—*SetState*.

The *SetState* activity, interestingly enough, isn't one of the activities you can drop into the *State* activity directly. Why is this? Because the *State* activity knows that something must trigger a state change. If you could drop *SetState* directly into *State*, when that state was entered it would immediately be transitioned out of. The state is then meaningless.

Instead, you drop an instance of *SetState* into two of the other three valid *State* child activities. Although it's more common to find *SetState* in *EventDriven* activities (changing to a new state as a result of an event), you will from time to time find it useful to drop *SetState* into *StateInitialization*. I do this in this chapter's sample application when the initial state merely sets things up for the remaining states. You cannot drop an instance of *SetState* into *StateFinalization*. *StateFinalization* is invoked when the particular state is being transitioned out of, so a previous instance of *SetState* would have already executed. It's too late to change your mind when *StateFinalization* is executing.

So aside from dropping an instance of *SetState* into your state's *EventDriven* or *StateInitialization* activities, how does it actually cause a transition to a different state? The answer is simply that *SetState*'s *TargetStateName* property should be set to the name of the state to be transitioned to.

Tip The visual workflow designer in Microsoft Visual Studio makes setting the *TargetStateName* easier by keeping track of all the states in your workflow and then presenting them to you in a drop-down list when you click on the *TargetStateName* property. For this reason, it's often better to drop all the states you'll require into your workflow before "wiring them up" using instances of *SetState*.

When the workflow runtime encounters a *SetState* activity when executing your workflow, it searches for the *State* activity bearing the same name as specified in the *TargetStateName* property. That state is then queued for execution. In the visual workflow designer, this is represented by an arrow traveling from the *SetState* activity to the state it indicates.

Using the *StateInitialization* Activity

When a state is transitioned into, you have the opportunity to initialize things related to that state through the *StateInitialization* activity. Although *StateInitialization* is not a required activity in your state, if an instance is present, WF guarantees that *StateInitialization* will be invoked before any other activity in your state.

StateInitialization derives from the *Sequence* activity, so nearly any activity you want to place in this composite activity is available to you and will be executed in sequential order. Certain activities are not allowed, such as any activity based on *IEventActivity*. To handle events in your state, you must use the *EventDriven* activity.

StateInitialization is also executed in a nonblocking manner. This approach is necessary because your state needs to be able to listen for events. If *StateInitialization* were a blocking activity, the thread executing the initialization code would be tied up and unable to listen for events. Note, however, that the event, although it has been received, will not be acted upon until *StateInitialization* completes. After all, critical initialization code might need to be executed prior to actually handling the event.

Using *StateInitialization*, and indeed any of the child activities in a given *State* activity, requires you to interact with the visual workflow designer in a slightly different way than you have so far in this book. If you drag and drop an instance of *StateInitialization* into a *State* activity, you'll find you can't then drop child activities directly into *StateInitialization*. (This is true for the *EventDriven* and *StateFinalization* activities as well.) To drop child activities into *StateInitialization*, you must first double-click the instance of *StateInitialization* you just dropped to activate the sequential workflow editor you've used in previous chapters. To return to the state-based workflow editor, you'll find hyperlink-style buttons in the upper-left corner of the workflow designer that will return you to a view of the particular state you're editing or the entire workflow.

Using the *StateFinalization* Activity

The *StateFinalization* activity is a mirror image of the *StateInitialization* activity and is used in a similar way. Where the *StateInitialization* activity is executed when the state itself begins execution, *StateFinalization* executes just prior to transitioning out of the state. Like *StateInitialization*, the *StateFinalization* activity is based on the *Sequence* activity. *StateFinalization* also limits the types of activities it can contain—*IEventActivity* activities are disallowed, as are *State* and *SetState*.

Creating a State-Based Workflow Application

If you recall the sample state machine I presented in Chapter 4, Figure 14-1 will look familiar. Yes, it's the (simplified) vending machine state diagram. I thought it might be interesting to build this state diagram into a true WF state-based workflow and then drive it using a user interface that, given my feeble artistic abilities, models a crude soft drink ("soda") vending machine.

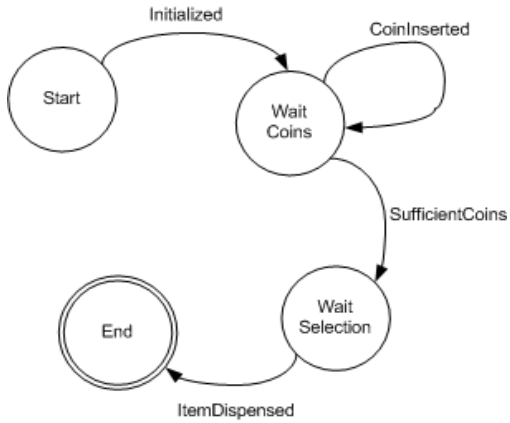


Figure 14-1 The SodaMachine state diagram

Given no user interaction, the soda machine application appears as you see in Figure 14-2. A bottle of soda costs $1.25. While you insert coins, the soda buttons on the left are inactive. However, when you insert enough money, the soda buttons are enabled and you can make a selection. This simplified model doesn't deal with such things as refunds and making change, but feel free to modify the application if you wish.

Note For the sake of simplicity, I did not internationalize the sample application. It simulates a vending machine that accepts United States currency only. However, keep in mind the salient point here is the workflow, not the monetary unit used.

Figure 14-2 The SodaMachine user interface in its initial state

However, you can't actually insert coins into a Windows Forms application, so I provided buttons for 5¢, 10¢, and 25¢. Sorry, coins only. When you click one of the coin buttons for the first time, a new state-based workflow instance is started, one that implements the workflow shown in Figure 14-1. Figure 14-3 shows you the soda machine after several coins have been added. The state-based workflow keeps track of the coinage received and reports the total back to the application, which displays it in the simulated liquid crystal diode (LCD) display.

Figure 14-3 The SodaMachine user interface as coins are added

When sufficient coins have been inserted, the workflow notifies the application that the user can now make a selection, as shown in Figure 14-4. The application, in turn, enables the individual soda buttons, located on the left of the user interface.

Figure 14-4 The SodaMachine user interface allowing soda selection

When one of the soda buttons on the left has been clicked, as the darkened button in Figure 14-5 indicates, a label appears that says "Soda!" This is my way of simulating the bottle of soda dropping from the machine for the customer to retrieve. To reset the entire process, click the Reset button. This doesn't affect the workflow but rather resets the user interface buttons. The user interface now appears as it did in Figure 14-2, and you can start the process all over again.

Figure 14-5 The SodaMachine user interface once a selection has been made

A great deal of the application has been created for you. If you wade through the sample SodaMachine code, you'll find I used the *CallExternalMethod* activity (from Chapter 8, "Workflow Data Transfer") and the *HandleExternalEvent* activity (from Chapter 10, "Event Activities"). These are great tools for interacting with your workflow from your application. What's left to create is the workflow itself, and here's how.

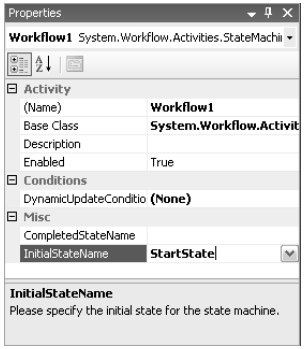
Building a state-based workflow

1. The SodaMachine application in its initial form can be found in the \Workflow\Chapter14\SodaMachine directory. As I've done in the past for the more complex sample applications, I placed two different versions in the Chapter14 directory—an incomplete version and a completed version. If you're interested in following along but don't want to perform the steps outlined here, open the solution file for the completed version. (It will be in the SodaMachine Completed directory.) The steps you'll follow here take you through building the state-based workflow. If you're interested in working through the steps, open the incomplete version instead. To open either solution, drag the .sln file onto a copy of Visual Studio to open the solution for editing and compilation. (If you decide to compile and execute the completed version directly, compile it twice before executing it. Internal project-level dependencies must be resolved after the first successful compilation.)
2. With the SodaMachine solution open in Visual Studio, press F6 or select Build Solution from Visual Studio's Build menu. The projects have various dependencies, and compiling the solution generates assemblies that dependent projects can reference.
3. Find the Workflow1.cs file in the SodaFlow project, within Visual Studio's Solution Explorer window. (You might need to expand tree control nodes to find it.) When the file is in view in the tree control, select it with a single mouse click and then click the View Designer toolbar button. This brings the workflow into the visual workflow designer for editing.

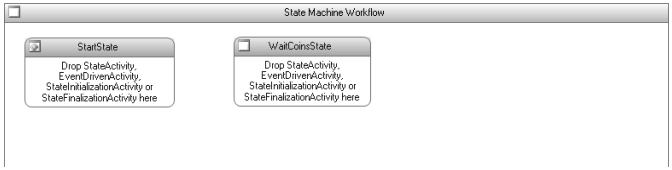
> **Note** I have already created the basic workflow project because the application uses *CallExternalMethod* and *HandleExternalEvent* activities using the techniques you saw in Chapters 8 and 10. There wasn't any need to rehash the steps necessary to create the custom activities, as you would need to do if you created the workflow project from scratch. (If you were starting from scratch you'd add a new workflow library project using the State Machine Workflow Library template.)

4. The workflow consists of a single *State* activity. Select the activity, *stateActivity1*, and change its name to **StartState**. You will find its *Name* property in Visual Studio's Properties pane when the activity is selected in the visual workflow designer.

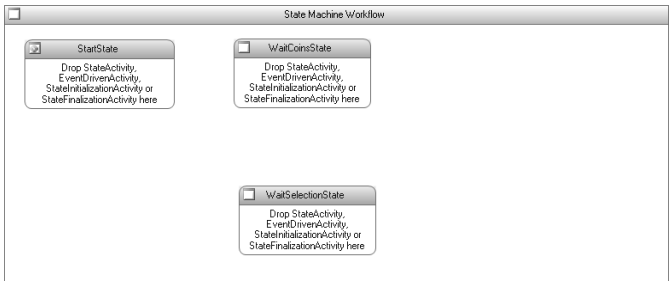
5. When the workflow was created, Visual Studio inserted this original *State* activity for you. But it also established this activity as the initial, or "start," activity. When you renamed the activity in the preceding step, the workflow lost this linkage. To reestablish this activity as the start activity, click once anywhere in the visual workflow designer's surface except for the *State* activity to activate the properties for the workflow as a whole. In the Properties pane, you should see an *InitialStateName* property. Change that from *stateActivity1* to **StartState**. Note you can either type this value into the property itself or use the drop-down list and select StartState.



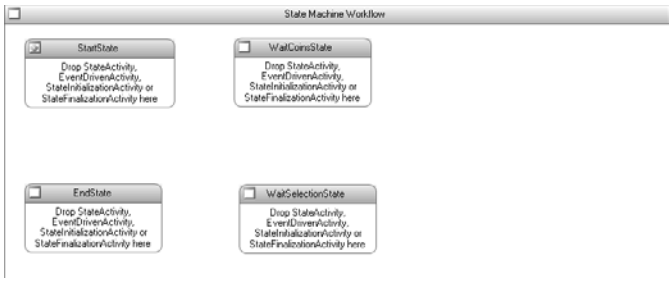
6. Let's now drop the remaining *State* activities onto the visual workflow designer's surface. As you recall, this facilitates assigning target states when working with *SetState*. From the Visual Studio Toolbox, drag an instance of the *State* activity onto the designer's surface and drop it next to the *StartState* activity. Change its name to **WaitCoinsState**.



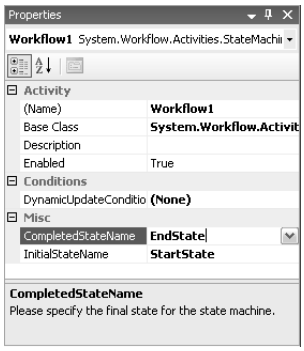
7. Drop another *State* activity onto the visual workflow designer's surface, and name it **WaitSelectionState**.



8. Drop the final *State* activity onto the visual workflow designer's surface, and change its name to **EndState**.

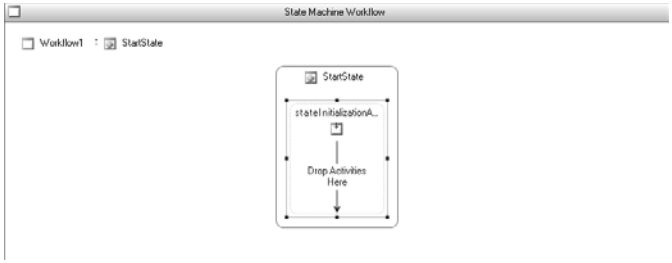


9. Just as you reassigned the starting state, so too will you need to tell WF what the ending state will be. Click the visual workflow designer's surface outside any *State* activity to enable the workflow properties. Assign the *CompletedStateName* property to be **EndState**. Visual Studio then clears *EndState*'s contents and changes the icon in the upper-left corner. As before, you can type **EndState** or select it from the drop-down list.

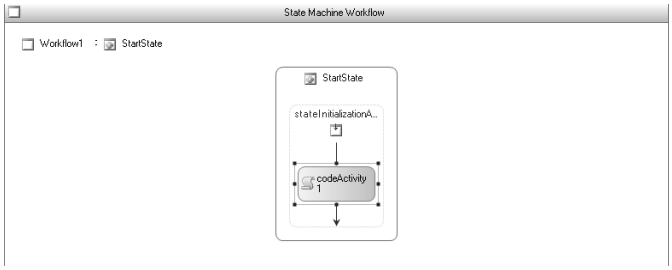


10. With the state activities in place, let's now add details. Starting with *StartState*, drag an instance of the *StateInitialization* activity from the Toolbox and drop it into *StartState*.

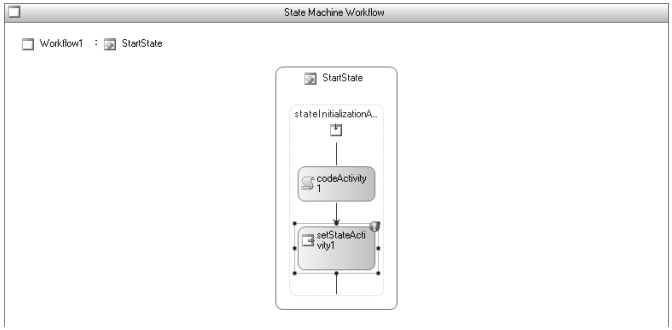
11. Double-click the activity you just inserted, *stateInitialization1*, to enter the sequential workflow editor.



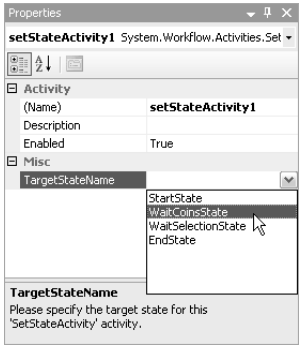
12. Drag a copy of the *Code* activity from the Toolbox, and drop it into the state initialization activity. Assign its *ExecuteCode* method to be **ResetTotal**. Visual Studio then adds the *ResetTotal* method for you and switches you to the code editor. Rather than add code at this point, return to the visual workflow designer.



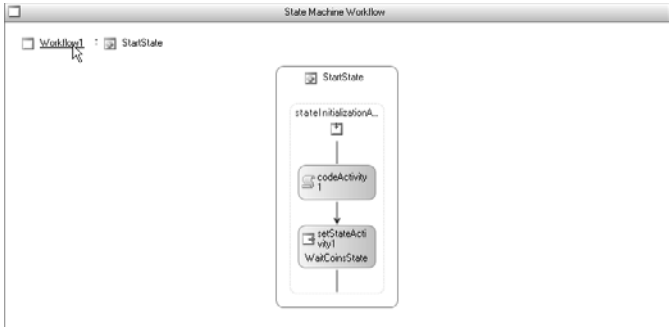
13. Next drag an instance of *SetState* onto the designer's surface, and drop it just below the *Code* activity you just inserted.



14. Assign the *SetState*'s *TargetStateName* property to be **WaitCoinsState**.

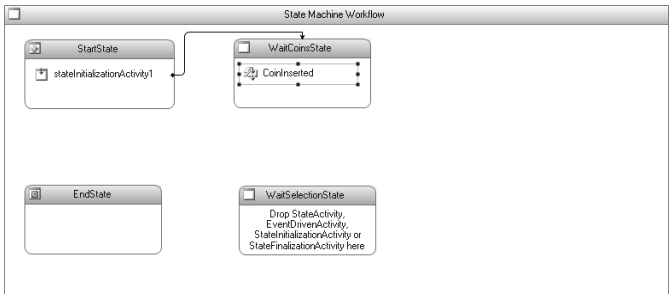


15. To return to the visual workflow designer's state editor view, click the Workflow1 hyperlink-style button in the upper-left corner.

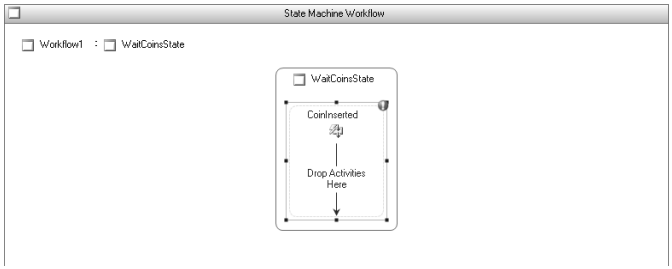


The state editor should now indicate that *StartState* transitions to *WaitCoinsState*.

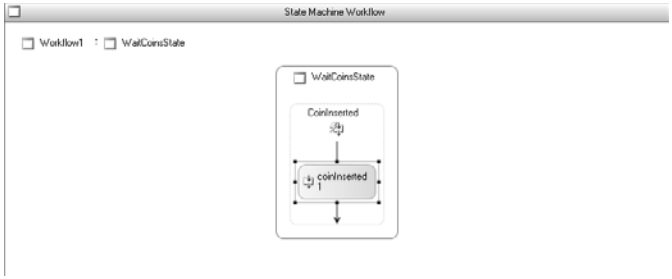
16. *StartState* is now complete. Next we'll turn to *WaitCoinsState*. To begin, drag a copy of the *EventDriven* activity onto the designer's surface and drop it into *WaitCoinsState*. Name it **CoinInserted** by changing its *Name* property in the Visual Studio Properties pane (you must press Enter for the change to take place).



17. Double-click the *CoinInserted EventDriven* activity to enable the sequential workflow editor.

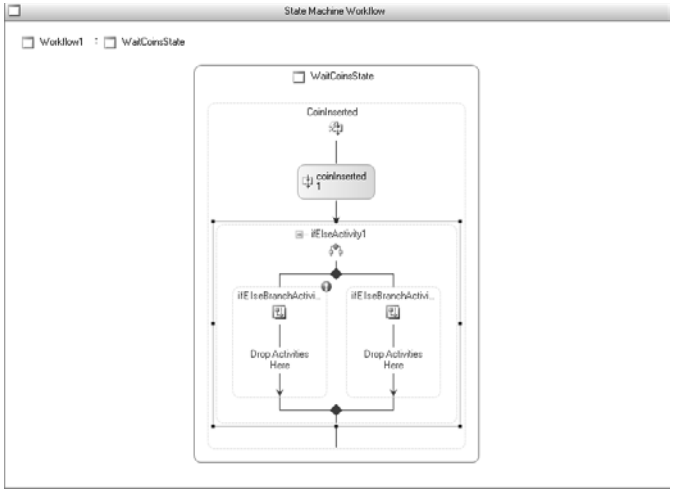


18. Now drag an instance of the *CoinInserted* custom activity from the Toolbox and drop it onto the *EventDriven* activity's surface. Note that if you haven't yet compiled the entire solution, the *CoinInserted* event doesn't appear in the Toolbox. You might have to remove the EventDriven activity to successfully compile if you skipped step 2.

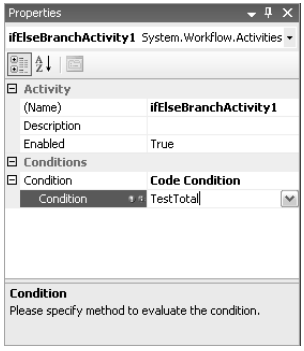


19. With the *ExternalEventHandler coinInserted1* activity selected in the visual workflow designer, click the *CoinValue* property in the Properties pane to activate the browse (...) button, and then click the browse button. This brings up the Bind 'CoinValue' To An Activity's Property dialog box. Click the Bind To A New Member tab, and type **LastCoinDropped** in the New Member Name field. The Create Property option should be selected, but if it isn't, select it so that you create a new dependency property. Click OK.

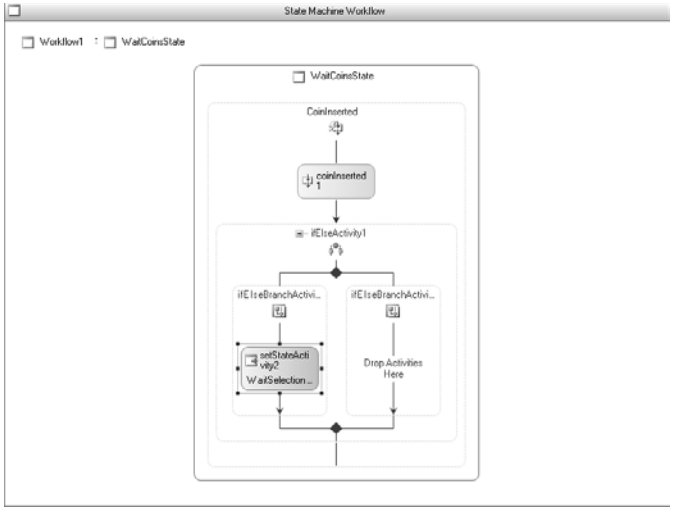
20. Now we need to make a decision—did the user just drop enough money to enable soda selection? To do this, drag an instance of the *IfElse* activity onto the visual workflow designer's surface and drop it into the *CoinInserted EventDriven* activity, following the *coinInserted1* event handler.



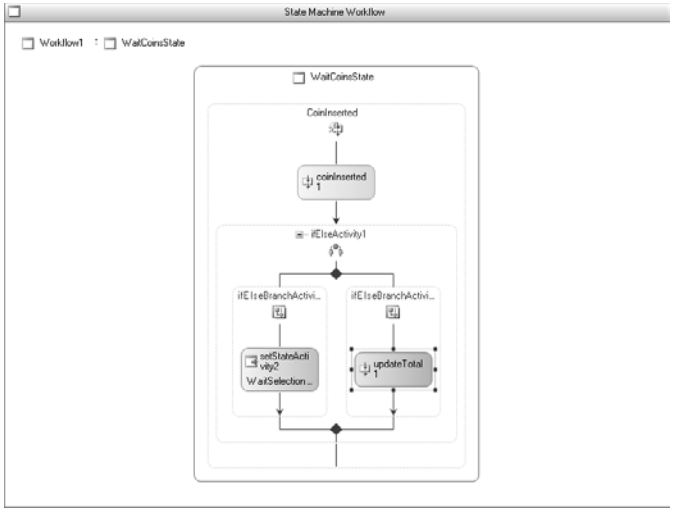
21. Select the left branch of *ifElseActivity1* to display its properties in the Properties pane. For its *Condition* property, select Code Condition. Expand the Condition node and in the child Condition property, type **TestTotal**. When Visual Studio adds the new method and switches you to the code editor, return to the visual workflow designer.



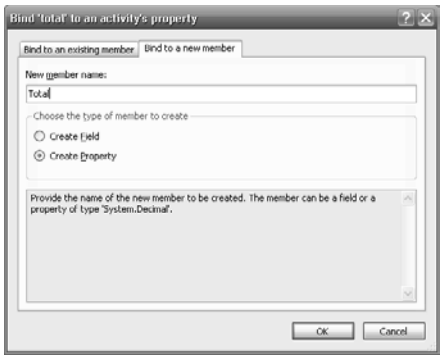
22. *TestTotal* will eventually check the total amount of money inserted into the soda machine. (We'll finish the workflow in the visual workflow designer before adding code because there are properties we need that have not yet been created.) If enough money has been inserted, we need to transition to the *WaitSelectionState*. Therefore, drag a copy of *SetState* into the left *IfElse* activity branch, *ifElseBranchActivity1*, and drop it. Assign its *TargetStateName* to be **WaitSelectionState**.



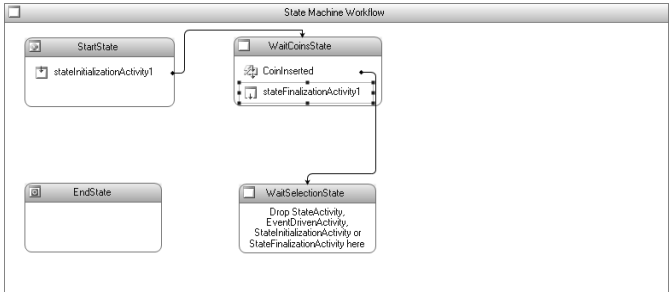
23. If *TestTotal* decides there isn't enough money to purchase a soda, the workflow needs to communicate the total amount of money inserted into the soda machine so far. To do this, drag an instance of *UpdateTotal* from the Toolbox and drop it into the right *IfElse* activity branch. *UpdateTotal* is a customized instance of *CallExternalMethod* I created for the job.



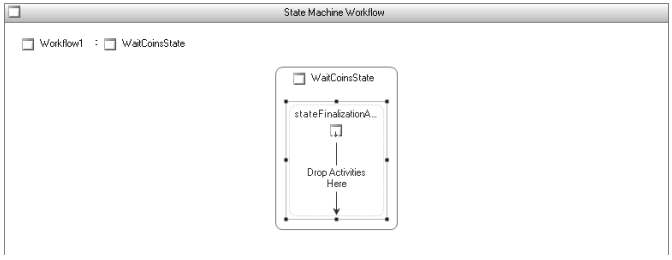
24. *UpdateTotal* requires a total value to communicate, so select its *total* property and click the browse (...) button to activate the bindings dialog box once again. When the bindings dialog box appears, select the Bind To A New Member tab and type **Total** into the New Member Name field, again making sure the Create Property option is selected. Click OK.



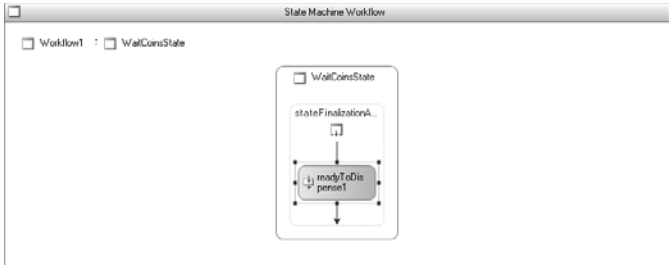
25. Click the Workflow1 hyperlink-style button in the upper-left corner to return to the state designer view. Drag an instance of *StateFinalization* onto the visual workflow designer's surface, and drop it into *WaitCoinsState*.



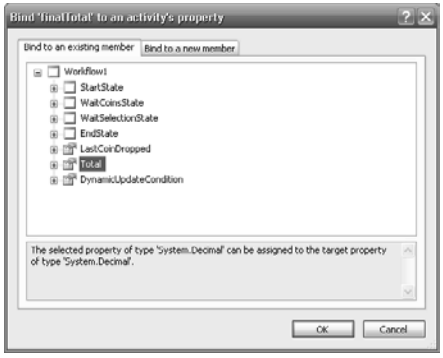
26. Double-click the *stateFinalizationActivity1* activity you just inserted to reactivate the sequential designer view.



27. From the Toolbox, drag an instance of *ReadyToDispense* and drop it into *stateFinalizationActivity1*. *ReadyToDispense* is also a customized *CallExternalMethod* activity.

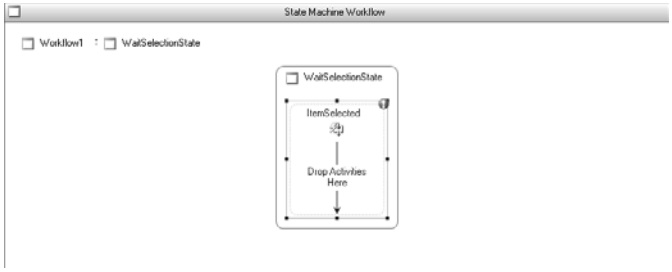


28. *ReadyToDispense1*, the activity you just inserted, will return the final total to the main application. To do that, it needs to access the *Total* property you inserted in step 24. Looking at *readyToDispense1*'s properties, click the *finalTotal* property, and then click the browse (...) button in the *finalTotal* property. Clicking the browse button activates the binding dialog box, but this time bind to an existing member. Select the *Total* property from the list and click OK.

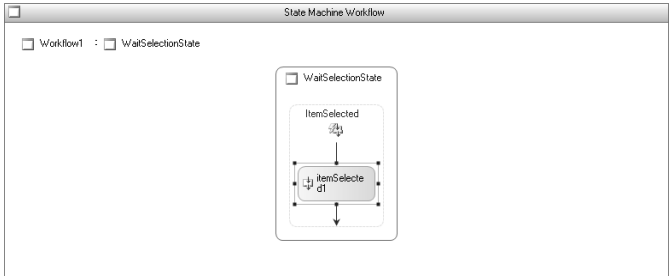


29. Click the Workflow1 hyperlink-style button to return to the state designer view. There, select the *EventDriven* activity from the Toolbox and drag it onto the designer's surface, dropping it into the *WaitSelectionState* activity. Name it **ItemSelected**.

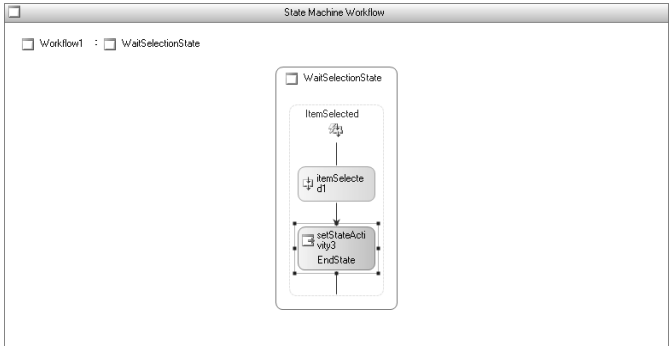
30. Double-click the *ItemSelected EventDriven* activity to enter the sequential designer view.



31. Drag a copy of the custom *ExternalEventHandler* activity *ItemSelected*, and drop it into the *ItemSelected EventDriven* activity.



32. After the user makes a selection, the main application fires the *ItemSelected* event. When that happens, we want to transition to *EndState*. To do that, of course, we need to insert a copy of the *SetState* activity. So drag an instance of *SetState* from the Toolbox and drop it into the *ItemSelected EventDriven* activity following the *itemSelected1* event handler. Assign its *TargetStateName* to be **EndState**.



33. Click the Workflow1 hyperlink-style button to return to the state designer view.

34. The workflow is complete from a visual workflow designer's point of view, but we still have some code to write. Select Workflow1.cs in Visual Studio's Solution Explorer, and click the View Code toolbar button to open the file for editing in the code editor.

35. Scan the Workflow1.cs source file, and locate the *ResetTotal* method you added in step 12. Insert the following code in the *ResetTotal* method:

```
// Start with no total.
Total = 0.0m;
```

36. Finally, locate the *TestTotal* method you added in step 21. To that method, add this code:

```
// Add the last coin dropped to the total and check
// to see if the total exceeds 1.25.
Total += LastCoinDropped;
e.Result = Total >= 1.25m;
```

37. Compile the entire solution by pressing F6 or by selecting Build Solution from Visual Studio's Build menu. Correct any compilation errors.

Now you can run the application by pressing F5 or Ctrl+F5. Click a coin button. Does the total update in the LCD display? When you insert enough money, can you select a soda?

Note If the application crashes with an *InvalidOperationException,* it's most likely due to the references not being fully updated by the first complete solution compilation. Simply recompile the entire application (repeat step 37) and run the application again. It should run cleanly.

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 15, "Workflows and Transactions." In Chapter 15, you'll take your first steps into the fascinating world of workflow transactional processing.

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and close it. Who needs transactions anyway? Actually, we all do, but we'll wait for you.

Chapter 14 Quick Reference

To	Do This
Add new states to your state-based workflow	Drag as many copies of the *State* activity onto the visual workflow designer's surface as you require. Remember it's easier to wire the states together (using the *SetState* activity) with the states in place. However, this is not a requirement.
Receive events within your workflow's states	Drag instances of *EventDriven* into your *State* activity, and assign event handlers to each event. *EventDriven* can accept only a single event, so you might need to drop multiple copies of the *EventDriven* activity into your *State* activity—one for each discrete event you need to accept.
Transition between states	Drag an instance of *SetState* activity into your state's *EventDriven* activity or *StateInitialization* activity. Assign the *TargetStateName* to the name of the state you want to transition to.
Initialize your state as it is transitioned into	Drag a copy of the *StateInitialization* activity into your *State* activity, and drop the necessary activities into *StateInitialization* as required for your initialization process. *StateInitialization* is a composite, sequential activity, but it will allow for events to be accepted by your state event handlers (even if the processing of those events is deferred until the initialization work is complete). Note that only a single instance of *StateInitialization* is allowed per *State* activity.
Execute code as your state is transitioned out of	Drag an instance of *StateFinalization* onto the visual workflow designer's surface, and drop it into your *State* activity. Like *StateInitialization*, the *StateFinalization* activity is a composite, sequential activity, and only one per *State* activity is allowed.