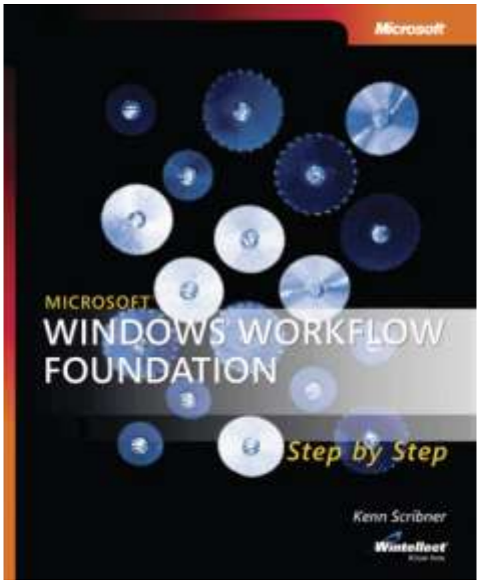


Microsoft® Windows® Workflow Foundation Step by Step

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

Table of Contents

Chapter 15

Workflows and Transactions

After completing this chapter, you will be able to:

- Understand the classical transaction model and where that model does and does not fit
- Know where classical transactions do not fit and when compensated transactions are appropriate
- See how transactions are rolled back or compensated
- See how to modify the default order of compensation

If you write software, sooner or later you'll need to understand transactional processing. *Transactional processing* in this sense means writing software that records information to a *durable resource*, such as a database, Microsoft Message Queue (which uses a database under the covers), Windows Vista with transacted file system and Registry access, or even some other software system that supports transactional processing. Durable resources retain the written information no matter what happens to them once the data has been recorded.

Transactions are critical to any business process because, by using transactions, you can be sure the data contained within your application is consistent. If the business process sustains an error yet still persists any data, the erroneous data most likely will propagate throughout the system, leaving you to question which data is good and which data is bad. Imagine ordering this book from an online merchant, only to find the merchant "had a little accident" with your credit card transaction and charged you 100 times the face value of the book instead of their discounted price. Transactional processing isn't a laughable or avoidable subject when errors such as this can happen.

Understanding Transactions

Transactional processing, at its very core, is all about managing your application's state. By *state*, I really mean the condition of all the application's data. An application is in a determinate state when all of its data is consistent. If you insert a new customer record into your database and that update requires two insertions (one to add a normalized row to tie the address to your customer and one to record the actual address information), adding the normalized row but failing to insert the address itself places your application in an indeterminate state. What will happen later when someone tries to retrieve that address? The system says the address should be there, but the actual address record is missing. Your application data is now inconsistent.

To be sure both updates are successful, a *transaction* comes into play. A transaction itself is a single unit of work that either completely succeeds or completely fails. That's not to say you can't update two different database tables. It just means that both table updates are considered a single unit of work, and both must be updated or else neither one is. If either or both updates fail, ideally you want the system to return to its state just prior to your attempt to update the tables. Your application should move forward with no evidence that there had been an incomplete attempt to modify the tables, and more important, you don't want to have data from the unsuccessful update in one table but not in the other.

Note Entire volumes have been written about transactions and transactional processing. Although I'll describe the concepts in sufficient depth to explain how Microsoft Windows Workflow Foundation (WF) supports transactions, I cannot possibly cover transactional processing in great depth in this book. If you haven't reviewed general transactional support in .NET 2.0, you should do so. WF transactions model .NET 2.0 transactional support very closely, and you might find the information in the following article helpful to understanding WF transactional support: *msdn2.microsoft.com/en-us/library/ms973865.aspx*.

Traditionally, transactions have come in a single form—that of the *XA*, or *two-phase commit*, style of transaction. However, with the advent of Internet-based communication and the need to commit long-running transactions, a newer style of transaction was introduced known as the *compensated* transaction. WF supports both styles. We'll first discuss the classical transaction, and then after noting the conditions that make this type of transaction a poor architectural choice, we'll discuss the compensated transaction.

Classic (XA) Transactions

The first system known to have implemented transactional processing was an airline reservation system. Reservations that required multiple flights could not progress if any of the individual flights could not be booked. The architects of that system knew this and designed a transactional approach that today we know as the. X/Open Distributed Transaction Processing Model, known as *XA*. (See *en.wikipedia.org/wiki/X/Open_XA*.)

An XA transaction involves the XA protocol, which is the two-phase commit I mentioned earlier, and three entities: the application, resource, and transactional manager. The application is, well, your application. The resource is a software system that is designed to join in XA-style transactions, which is to say it *enlists* (joins) in the transaction and understands how to participate in the two phases of committing data as well as provides for durability (discussed shortly). The transactional manager oversees the entire transactional process.

So what is a two-phase commit? In the end, imagine your application needs to write data to, say, a database. If that write is performed under the guise of a transaction, the database holds the data to be written until the transactional manager issues a *prepare* instruction. At that point, the database responds with a *vote*. If the vote is to go ahead and commit (write) the data into a table, the transaction manager proceeds to the next participating resource, if any.

If all resources vote to commit the data, the transactional manager issues a *commit* instruction and each resource writes the data into its internal data store. Only then is the data destined for your table actually inserted into the database.

If any one resource has a problem and votes not to commit the data, the transactional manager issues a *rollback* instruction. All resources participating in the transaction must then destroy the information related to the transaction, and nothing is permanently recorded.

Once the data has been committed, the XA protocol guarantees that the result of the transaction is permanent. If data was inserted, it is there for your application to use. If information was deleted, it has been deleted permanently. Your application, then, can move forward comfortable in the knowledge that all is well with the data. The data is consistent, and the application is in a determinate state.

ACID Properties

When we speak of XA transactions, it's hard not to mention the ACID acronym—Atomic, Consistent, Isolated, and Durable (*en.wikipedia.org/wiki/ACID*). All XA-style transactions, to non-volatile resources, must exhibit these properties or the transaction is architecturally invalid.

By *atomic*, we mean the resource enlisted in the transaction supports the two-phase commit protocol. The data to be transacted is either completely transacted (updated, deleted, or whatever) or none of it is. If the transaction fails, the resource returns to the state just prior to the attempt to transact the data.

Consistency means the data maintains integrity. For databases, this typically means the data doesn't violate any constraints, but for other resources maintaining integrity might have different or additional connotations. If the data violates any rules or constraints, which ultimately would result in an indeterminate application state, the resource must vote to roll back the transaction to prevent inconsistent data from being permanently recorded in the system.

Isolation is the transactional property that causes the system to be unable to access data while a transaction is ongoing. In a database, attempting to write to a previously locked row, or perhaps reading from a row with uncommitted data, is disallowed. Data is available only when it has been committed, or in the case of the read operation, when you explicitly allow uncommitted reads (often called "dirty reads").

Durable resources guarantee that when the data is committed it will always be available in a nonvolatile manner. If the data is committed and the power to the database server is cut off one millisecond later, when the database server is back online that data will be in the database, ready for your application to use. This is much more difficult to do in practice than it sounds, and it is one of the primary reasons architects use a database for persistent data storage rather than simple data files, such as XML, for critical data. (Admittedly, Windows Vista might change things a bit with its transacted file system, but hopefully you see my point.)

Long-Running Transactions and Application State

Keep in mind that the entire premise of the XA-style transaction is that your application will retain its original state if the transaction rolls back. But consider this: What happens to your application if a transaction takes an inordinate amount of time to commit?

Before I answer that, imagine your online purchasing system received an order from a customer, but the credit card validation process got hung up. Clearly your process is running within a transaction because you don't want to charge the customer if something fails. But in the meantime, other customers are placing orders. Lots of orders, if you're fortunate. If the first customer's transaction later fails, what will happen to the orders placed in the meantime?

If the system isn't designed to isolate individual order failures, then the correct thing to do is to roll the system completely back to its original state. But considering this, that means we not only lose the first customer's order, but we also lose *every other* customer's order that was placed in the interim. Even if it's only two orders, that's not good. But if it's 10,000 orders…the loss of that amount of revenue can't be tolerated.

Of course, we'll retain those 10,000 orders and just deal with the first customer as an isolated event, but we're taking a chance in this case and intentionally breaking one of the four transactional properties to retain the revenue. It's a calculated risk, but often a risk we must accept in real-world situations.

The property that's being broken is actually atomicity, and for this reason people who write transactional processing systems strive to keep their transactions as short as possible. You do only what is required within your transactional bounds and no more, and you do so as efficiently as possible so that the transaction completes quickly.

Now let's throw in another complication—the Internet. Your customer is ordering online, and networks are notorious for slow speeds and even disconnections. So transactional processing over the Internet is questionable if only because sooner or later a transaction will run overlong and put our online ordering system in a transactional bind.

Compensation as a Solution

It is precisely this situation that created the need for a *compensated* transaction. If I give you five apples using an XA-style transaction and the transaction fails, time itself rewinds to the point I started to give you the apples. In a sense, history is rewritten such that the five apples were never given in the first place. But if I give you five apples in a compensated transaction and that transaction fails, to compensate (so that we maintain a determinate application state), you must return five apples to me. It might seem like a subtle difference, but there is a definite difference between the two styles of transactions.

When writing XA-style transactions, the responsibility for rolling back failed transactions falls to the resource, such as your database. Conversely, when a compensated transaction fails, you—as a transactional participant—are responsible for compensating by providing a compensation function for your part of the transaction. If you debited an online consumer's credit card and were later told to compensate, you would immediately credit the customer's account with the same amount of money you originally debited. In an XA-style transaction, the account would never have been debited in the first place. With the compensated transaction, you initiate two actions–one to debit the account and one to later credit it.

Note Make no mistake, it would be a rare system that could successfully perform XA-style transactions over the Internet. (I would argue that no system can, but I would be doing just that—starting an argument—so I accept the fact that some systems will try and even succeed in some cases.) Compensation is generally called for. But craft your compensation functions very carefully. Pay attention to details. If you don't, you could be making a bad situation worse by injecting error upon error. It is often not easy to write accurate compensation functions.

Initiating Transactions in Your Workflows

In general, initiating transactions in WF is as simple as dropping a transaction-based activity into your workflow. If you're using transactional activities, however, there is a little more you should know.

Workflow Runtime and Transactional Services

When you use a transaction-based activity in your workflow, two workflow-pluggable services are required. First, because the two out-of-the-box transaction-based WF activities are both decorated with the *PersistOnClose* attribute (mentioned in Chapter 6, "Loading and Unloading Instances"), you must also start the *SqlWorkflowPersistenceService*. If you do not, WF won't crash, but neither will your transactions commit.

Perhaps more interesting for this chapter is the *DefaultWorkflowTransactionService* that WF starts on your behalf when the workflow runtime is started. This service is responsible for both starting and committing your transactional operations. Without such a service, transactions within the workflow runtime are not possible.

Note Although it's beyond the scope of this chapter, you can create your own transactional services. All WF transactional services derive from *WorkflowTransactionService*, so creating your own service is a matter of overriding the base functionality you want to change. In fact, WF ships with a customized transactional service for shared Microsoft SQL Server connections, *SharedConnectionWorkflowTransactionService*. You can find more information at *msdn2.microsoft.com/en-us/library/ms734716.aspx*.

Fault Handling

Although it isn't required that you handle faults in your workflow due to transactional failures, it's good practice. But I don't mention it here simply because it could be considered a best practice. I mention it because it is possible for you to write your own transactional service that automatically examines the exception and retries the transaction before actually failing. Although demonstrating how to do this is outside the scope of this chapter, you should know this is possible.

Ambient Transactions

The transaction-based activities all work with something known as the *ambient transaction*. When your workflow enters a transactional scope, the workflow transactional service automatically creates a transaction for you. There is no need to try and create one yourself. The activities embedded in a transactional scope all belong to this one ambient transaction and are committed or rolled back (or compensated) if the transaction succeeds or fails.

Using the *TransactionScope* Activity

XA-style transactions in WF are implemented by the *TransactionScope* activity. This activity is closely aligned with the .NET *System.Transactions* namespace, and in fact it initiates a *Transaction* as the ambient transaction when the activity begins execution. The *TransactionScope* activity even shares data structures (*TransactionOptions*) with *System.Transactions*.

Using the composite activity-based *TransactionScope* is truly as easy as dropping it into your workflow. Any activity you place inside the *TransactionScope* activity automatically inherits the ambient transaction and operates as typical transactions do when using .NET's own *System.Transactions*.

Note You cannot place a *TransactionScope* activity within another transactional activity. Nesting of transactions is not permitted. (This rule holds true for *CompensatableTransactionScope* as well.)

Transactional options dictate more precisely how the ambient transaction will operate. These options, supported by the *System.Transactions.TransactionOptions* structure, allow you to set the isolation level and timeout that the ambient transaction will support. The timeout value is self-explanatory, but the isolation level might not be.

Note The timeout values have limits, which are configurable. There is a machine-wide setting, *System.Transactions.Configuration.MachineSettingsSection.MaxTimeout*, and a local one, *System.Transactions.Configuration.DefaultSettings.Timeout*, which set the ceilings on the maximum value to allow for a timeout. These values override anything you set using *TransactionOptions*.

A transaction's isolation level defines to a large extent what the transaction can do with data to be transacted. For example, maybe you want your transaction to be able to read uncommitted data (to preclude being locked out by a previous transactional database page lock). Or the data you are writing might be critical, and therefore you allow the transaction to read only committed data, and moreover, you disallow other transactions to work with the data while your transaction is executing. The isolation levels you can select are shown in Table 15-1. You set both the isolation level and timeout using the *TransactionOptions* property of the *TransactionScope* activity.

Table 15-1 Transactional Isolation Levels

Isolation Level	Meaning
Chaos	Uncommitted and pending changes from transactions using higher isolated level cannot be overwritten.
ReadCommitted	Uncommitted data cannot be read during the transaction, but it can be modified.
ReadUncommitted	Uncommitted data can be both read and modified during the transaction. However, keep in mind that the data may change—there is no guarantee that the data will be the same on subsequent reads.
RepeatableRead	Uncommitted data can be read but not modified during the transaction. However, new data can be inserted.
Serializable	Uncommitted data can be read but not modified, and no new data can be inserted during the transaction.
Snapshot	Uncommitted data can be read. But prior to the transaction actually modifying the data, the transaction verifies that another transaction has not changed the data after it was initially read. If the data has been changed, the transaction raises an error. The purpose of this is to allow a transaction to read the previously committed data value.
Unspecified	A different isolation level from the one specified is being used, but the level cannot be determined for some reason. If you try to set the transactional isolation level to this value, an exception is thrown. Only the transactional system can set this value.

When you drop an instance of the *TransactionScope* activity into your workflow, the isolation level is automatically set to *Serializable*. Feel free to change this as your architecture dictates. *Serializable* is the strictest isolation level, but it also limits scalability to some degree. It's not uncommon to select *ReadCommitted* as the isolation level for systems that require a bit more throughput, but this is a decision only your system can dictate based on your individual requirements.

Committing Transactions

If you're used to working with SQL Server transactions, or perhaps COM+ transactions, you know that once the data has been inserted, updated, or deleted you must commit the transaction. That is, you initiate the two-phase commit protocol and the database permanently records or removes the data.

However, this is not necessary with the *TransactionScope* activity. If the transaction is successful (no errors while inserting, updating, or deleting the data), the transaction is automatically committed for you when the workflow execution leaves the transactional scope.

Rolling Back Transactions

How about rolling back failed transactions? Well, just as transactions are committed for you, so too will the data be rolled back if the transaction fails. What is interesting about this is the rollback is silent, at least as far as WF is concerned. If you need to check the success or failure of your transaction, you need to incorporate logic for doing so yourself. *TransactionScope* doesn't automatically throw an exception if the transaction fails. It merely rolls back the data and moves on.

Using the *CompensatableTransactionScope* Activity

If an XA-style transaction won't do, you can instead drop the *CompensatableTransactionScope* activity into your workflow and provide for compensated transactional processing. The *CompensatableTransactionScope* activity, like *TransactionScope*, is a composite activity. However, *CompensatableTransactionScope* also implements the *ICompensatableActivity* interface, which gives it the ability to compensate for failed transactions by implementing the *Compensate* method.

Also like *TransactionScope*, the *CompensatableTransactionScope* activity creates an ambient transaction. Activities contained within *CompensatableTransactionScope* share this transaction. If their operations succeed, the data is committed. However, should any of them fail, you generally initiate the compensation by executing a *Throw* activity.

Tip Compensated transactions can enlist traditional resources, such as databases, and when the transaction commits, the data is committed just as if it were an XA-style transaction. However, a nice feature of compensated transactions is that you do not have to enlist an XA-style resource to store data. Sending data to a remote site using a Web service is the classic example for a nonenlistable transactional resource. If you send data to the remote site but later must compensate, you need to somehow communicate with the remote site that the data is no longer valid. (How you accomplish this depends on the individual remote site.)

Throw causes the transaction to fail and calls into execution your compensation handler for your *CompensatableTransactionScope* activity. You access the compensation handler through the Smart Tag associated with the *CompensatableTransactionScope* activity in much the same way you would add a *FaultHandler*.

Note Although throwing an exception kicks off the transactional compensation, the *Throw* activity itself is not considered handled. You can also decide to place a *FaultHandler* activity in your workflow to preclude premature workflow termination.

Using the *Compensate* Activity

When you are compensating a failed transaction implemented by *CompensatableTransaction-Scope*, the compensation handler is invoked. If you have multiple compensatable transactions, the transactions are compensated in a default order, starting with the deepest nested transaction and working outward. (You'll see how this might be accomplished in the next section.) When your logic calls for compensation, you can place a *Compensate* activity in your compensation handler to initiate compensation of all completed activities supporting *ICompensatableActivity*.

It will always be the case that exceptions will cause compensation, so the use of the *Compensate* activity is not required. Why have it then? Because you might have nested more than a single compensatable transaction in a *CompensatableSequence* activity. If one transaction fails and is to be compensated, you can initiate the compensation of the other transaction even if that transaction previously completed successfully.

Note The *Compensate* activity is valid only in compensation handlers, cancellation handlers, and fault handlers.

You should use the *Compensate* activity only when you need to compensate activities in an order other than the default compensation order. Default compensation invokes compensation for all nested *ICompensatableActivity* activities in the reverse order of their completion. If this ordering doesn't fit your workflow model, or if you want to selectively invoke compensation of completed compensatable child activities, the *Compensate* activity is the tool of choice.

Note The *Compensate* activity uses its *TargetActivityName* property to identify which compensatable activity should be compensated. If more than one compensatable activity should be queued for compensation, you need to use more than one *Compensate* activity. If you decide not to compensate a given transaction, simply do nothing in the compensation handler for that transaction or in the enclosing parent activity.

The *Compensate* activity provides you control over the compensation process by allowing you to decide whether you want to compensate an immediate child activity that supports compensation or not. This ability enables your workflow to explicitly perform compensation on a nested compensatable activity according to your process's needs. By specifying which compensatable activity you want to be compensated in the *Compensate* activity, any compensation code in that compensatable activity will be executed as long as the compensatable activity previously successfully committed.

If you want to compensate more than one nested compensatable activity, you add a *Compensate* activity in your handler for each compensatable activity you want to compensate. If the *Compensate* activity is used in a handler of a compensatable activity that contains embedded compensatable activities, and if *TargetActivityName* for that *Compensate* activity is assigned to the parent activity, compensation in all child (compensatable) activities that committed successfully is invoked. Try saying that three times, fast.

Using the *CompensatableSequence* Activity

The preceding section might leave you wondering why the *Compensate* activity exists. After all, you can't nest compensated transactions. You can't nest any type of WF-based transaction.

But let's look at it in a different way. How would you tie two compensatable transactions together so that the failure of one triggers compensation in the other, especially if the other already completed successfully? The answer is you pair the compensated transactions in a single instance of the *CompensatableSequence* activity. Then, in the compensation or fault handler for the *CompensatableSequence* activity, you trigger compensation of both child transactional scope activities if either one of them fails. Even more interesting is the situation where you tie three compensatable transactions together in a single *CompensatableSequence* activity and allow one transaction to succeed even if the others fail and are compensated. The *Compensate* activity gives you this control.

This highlights the intent of the *CompensatableSequence* activity. The *CompensatableSequence* activity, at its core, is a *Sequence* activity, and you use the *CompensatableSequence* activity in the same way you would any sequential activity. The major difference is that you can embed multiple compensatable activities in a single *CompensatableSequence* activity, effectively tying related transactions together. Coupling the *CompensatableSequence* activity with both the *CompensatableTransactionScope* and *Compensate* activities provides you with powerful transactional control in your workflow.

Note *CompensatableSequence* activities can be embedded within other *CompensatableSequence* activities, but they cannot be children of *CompensatableTransactionScope* activities.

Tip When combining multiple compensatable transactions in a single compensatable sequence, you do not have to assign compensation functions to the individual transacted activities. Compensation flows to the parent activity if called for, so you can collect your compensation activities in the enclosing compensatable sequence activity if you want to.

Creating a Transacted Workflow

I've created an application that simulates an automated teller machine (ATM), one where you provide your personal identification number, or PIN as it's called, and make deposits to or withdrawals from your bank account. Deposits will be embedded in an XA-style transaction, while withdrawals will be compensated if the action fails. To really exercise the transactional nature of the application, I placed a "force transactional error" check box in the application. Simply select the check box and the next database-related operation will fail.

The workflow for this application is a state-based one, and it is more complex than the application you saw in the previous chapter (Chapter 14, "State-Based Workflows"). I've shown the state machine I based the workflow on in Figure 15-1. Most of the application has already been written for you. You'll add the transactional components in the exercises to follow.

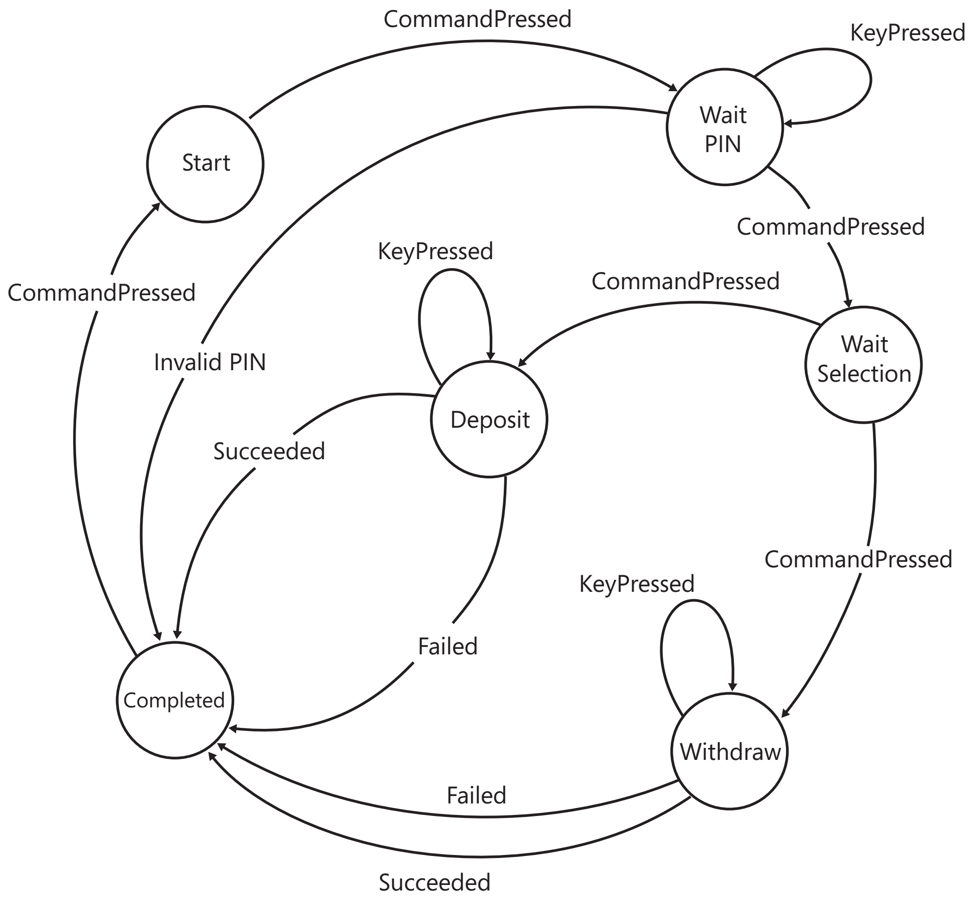
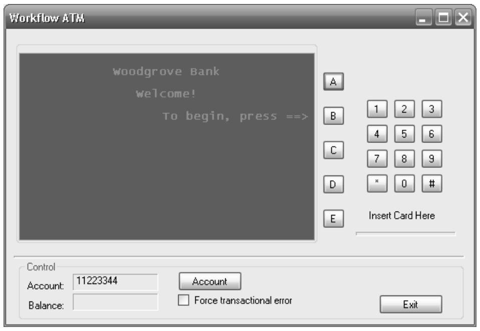


Figure 15-1 The WorkflowATM state diagram

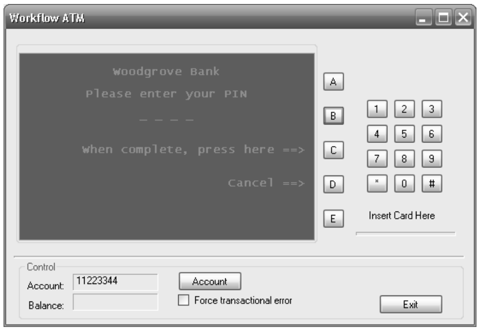
The user interface for the application is shown in Figure 15-2. This is the initial application state, akin to the ATM's state prior to inserting your bank card. Clearly, the sample can't deal with a true bank card, so clicking the B key transitions the user interface (and application state) to the PIN verification state (shown in Figure 15-3).



```
Woodgrove Bank
Welcome!
To begin, press ==>
```

Account: 11223344
Balance:

Figure 15-2 The WorkflowATM initial user interface

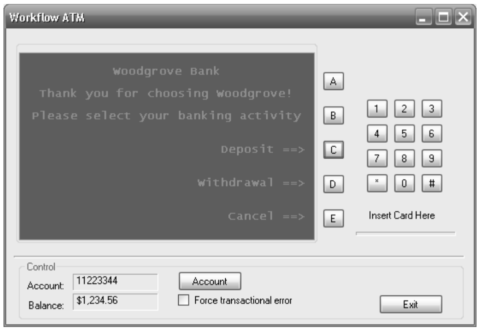


```
Woodgrove Bank
Please enter your PIN
_ _ _ _

When complete, press here ==>

Cancel ==>
```

Account: 11223344
Balance:

Figure 15-3 The WorkflowATM PIN verification user interface

You enter your PIN using the keypad to the right. Once the four-digit code is entered, you click the C key to kick off a database query to verify the PIN. If the PIN is verified (and note the account number in the lower-left corner; the PIN must be valid for that account number), the user interface transitions to the activity selection state, shown in Figure 15-4. Here you decide to either deposit funds to or withdraw funds from your account.

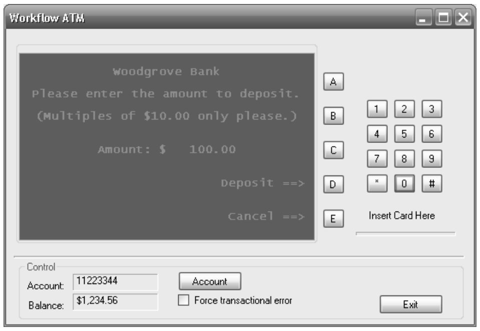


```
Woodgrove Bank
Thank you for choosing Woodgrove!
Please select your banking activity
Deposit ==>
Withdrawal ==>
Cancel ==>
```

Account: 11223344
Balance: $1,234.56
☐ Force transactional error

Figure 15-4 The WorkflowATM activity selection user interface

The application user interface for depositing and withdrawing funds is similar, so I've shown only the deposit user interface in Figure 15-5. You again use the keypad to enter a monetary value and then click a command key, the D key, to make the deposit or withdrawal or the E key to cancel the transaction.

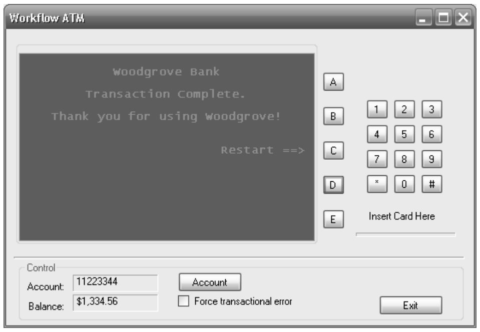


```
Woodgrove Bank
Please enter the amount to deposit.
(Multiples of $10.00 only please.)
Amount: $   100.00
Deposit ==>
Cancel ==>
```

Account: 11223344
Balance: $1,234.56
☐ Force transactional error

Figure 15-5 The WorkflowATM transaction deposit user interface

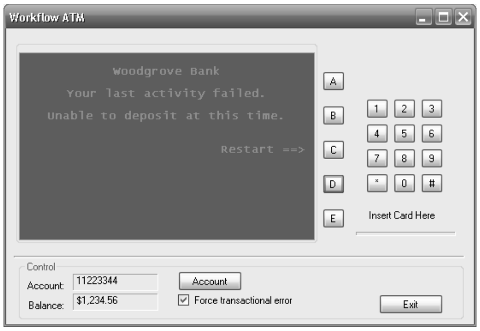
If the transaction was successful, you are rewarded with the screen you see in Figure 15-6. If not, you see the error screen shown in Figure 15-7. Either way, clicking the C key starts the workflow over again.



```
Woodgrove Bank
Transaction Complete.
Thank you for using Woodgrove!
Restart ==>
```

Account: 11223344
Balance: $1,334.56
☐ Force transactional error

Figure 15-6 The WorkflowATM transaction successful user interface



Woodgrove Bank
Your last activity failed.
Unable to deposit at this time.
Restart ==>

Account: 11223344
Balance: $1,234.56
[✓] Force transactional error

Figure 15-7 The WorkflowATM transaction failed user interface

The application requires a database to fully test WF's transactional capabilities. Therefore, I created a simple database used to store both user accounts with PINs and account balances. Several stored procedures are also available to help with the database interactions. All the stored procedures that involve a database update are required to execute within a transaction—I check *@@trancount*, and if it is zero, I return an error from each stored procedure. What this should prove is that the ambient transaction is being used if I fail to provide any ADO.NET code to initiate my own SQL Server transaction. What this also means is you need to create an instance of the database, but that's easily accomplished because you've learned how to execute queries in SQL Server Management Studio Express in previous chapters. In fact, let's start with that task because we'll soon need the database for application development and testing.

Note Before I forget to mention it, the database creation script creates a single account, 11223344, with the PIN 1234. The application allows you to change accounts and provide any PIN value you like, but unless you use this account (11223344) and this PIN (1234), or create your own account record, you will not be authorized to make deposits or withdrawals.

Creating the Woodgrove ATM databases

1. You should find the *Create Woodgrove Database.sql* database creation script in the \Workflow\Chapter15 directory. First find it and then start SQL Server Management Studio Express.

   Note Keep in mind that the full version of SQL Server will work here as well.

2. When SQL Server Management Studio Express is up and running, drag the Create Woodgrove Database.sql file from Windows Explorer and drop it onto SQL Server Management Studio Express. This opens the script file for editing and execution.

Note If SQL Server Management Studio Express requests a new connection to your database engine, make the connection and continue. The "Creating a SQL Server 2005 tracking database" procedure from Chapter 5, "Workflow Tracking," describes this process in detail if you need a refresher.

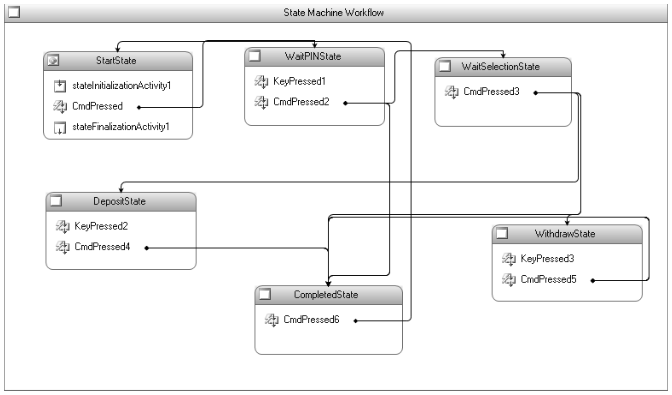
3. The script both creates the Woodgrove database and populates it with data. If you did not load SQL Server in the default directory, C:\Program Files\Microsoft SQL Server, you might need to edit the creation script to change the directory in which the database will be created. The fifth and seventh lines of the creation script indicate the database's directory and filename. Feel free to modify those as required to work within the bounds of your system. In most cases, you should not need to make changes. Click the Execute button to run the script and create the database. (You do not need to specify which database the query will run against since it creates an entirely new database.)
4. While you're using SQL Server Management Studio Express, if you didn't already work through the steps indicated in Chapter 6, "Loading and Unloading Instances," in the section "Setting Up SQL Server for Persistence," to install the workflow persistence database, do so now.

After completing these four steps, you'll have two databases ready to use: the Woodgrove database for banking information and the WorkflowStore database for workflow persistence. Now let's write some transacted workflow code.

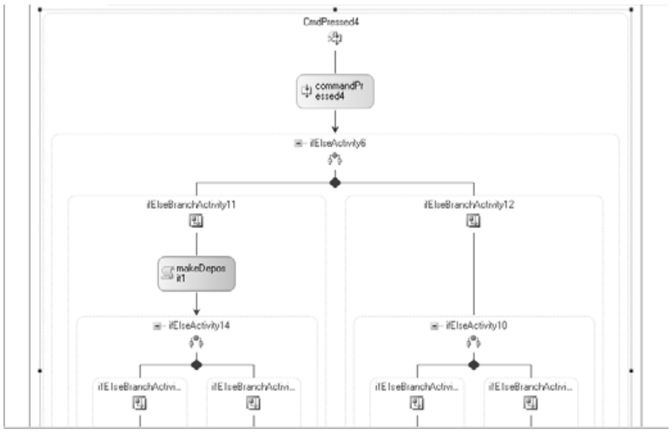
Adding an XA-style transaction to your workflow

1. You should find the WorkflowATM application in the \Workflow\Chapter15\WorkflowATM directory. As usual, I placed two different versions in the Chapter15 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 WorkflowATM Completed directory.) If you're interested in working through the steps, open the WorkflowATM version instead. To open either solution, drag the .sln file onto an instance of Microsoft Visual Studio to open the solution for editing and compilation. For either version of the sample application, you might need to change the connection strings in the App.Config file to match your SQL Server installation.
2. So that the custom activities show up in the Visual Studio Toolbox, press F6 to compile the entire solution. You can, alternatively, choose Build and then Build Solution from Visual Studio's main menu. The application will compile without error.
3. Although the WorkflowATM application is moderately complex, it follows the pattern we've used throughout the book. The Windows Forms application itself communicates with the workflow via a local communication service, using custom activities I created with *wca.exe*. The service is housed in the BankingService project, while the workflow is maintained in the BankingFlow project. The only code we'll concentrate on is in the

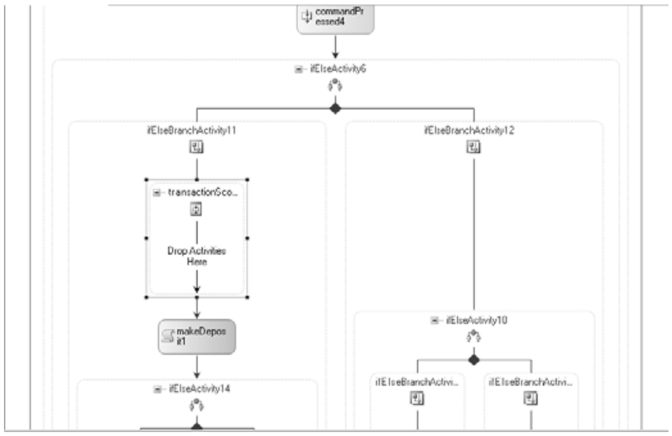
workflow itself. Locate the Workflow1.cs file in the BankingFlow project, and double-click it to open it for editing in the visual workflow designer. The workflow should appear as you see here. Does it look somewhat like Figure 15-1?



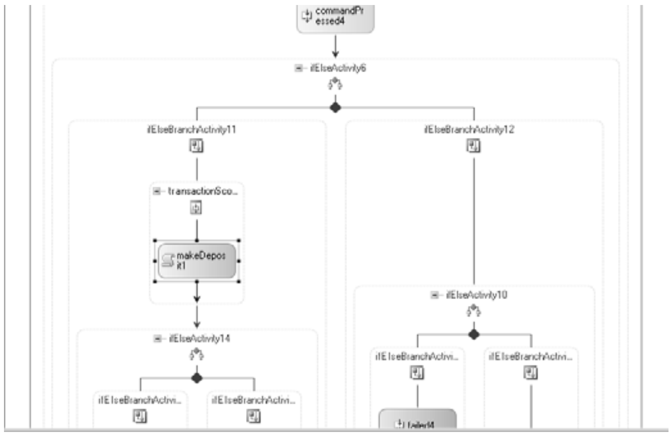
4. To insert the XA-style transaction, first double-click the *CmdPressed4 EventDriven* activity in the *DepositState* activity. This opens the *CmdPressed4* activity for editing.



5. Looking to the left, you should see the *Code* activity named *makeDeposit1*. Between this *Code* activity and the *ifElseBranchActivity11* title above *makeDeposit1*, drag an instance of the *TransactionScope* activity from the Toolbox and drop it.



6. Drag *makeDeposit1* from below the transaction scope activity you just inserted, and drop it inside so that the *makeDeposit1 Code* activity will execute within the transactional scope.



Note Feel free to examine the code contained in the *MakeDeposit* method, which is bound to the *makeDeposit1* activity. The code you find there is typical ADO.NET database access code. An interesting thing to see is that no SQL Server transaction is initiated in the code. Instead, the ambient transaction will be used when the code is executed.

7. Compile the entire solution by pressing F6 or by selecting Build Solution from the Visual Studio Build menu.

8. To test the application, press F5 or select Start Debugging from Visual Studio's Debug menu. The account should already be set. Click the B key to access the PIN verification screen, and then type **1234** (the PIN). Click the C key to verify the PIN and proceed to the activity selection screen.

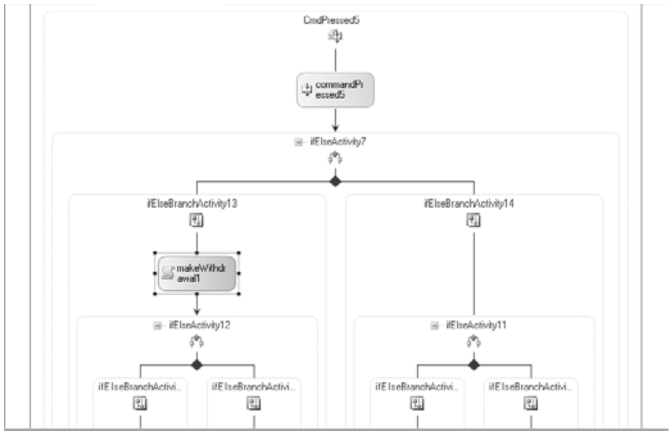
> **Note** If the application fails to verify the PIN, assuming you typed the correct PIN into the application, it might be because the connection string for the Woodgrove database is not correct. (The error handling is such that the application should not crash.) Verify that the connection string is correct, and run the application again. Chapter 5 has some suggestions for building connection strings.

9. Because you added the transaction to the deposit logic, click the C key to make a deposit.
10. Type **10** to deposit $100 (10 multiples of $10.00) and then click the D key to initiate the transaction. The transaction should succeed, and the screen now indicates the transaction is complete. Because the Woodgrove database creation script loaded the fictitious bank account with $1234.56, the balance now indicates $1334.56. Note you can read the balance in the lower-left corner of the application. Click the C key to return to the starting screen.
11. Now let's force the transaction to fail. The Deposit stored procedure takes as a parameter a value that causes the stored procedure to return an error. Selecting the Force Transactional Error check box assigns a value that causes the Deposit error. So click the B key to access the PIN verification screen yet again, then type **1234**, and then click the C key to access the banking activity selection screen.
12. Again, click the C key to make a deposit and enter **10** to deposit another $100, but this time select the Force Transactional Error check box *before* clicking the D key.
13. After clicking the D key, the application indicates a transactional failure, but notice the balance. It indicates the current balance is still $1334.56, which was the balance prior to the transaction. Both the successful transaction (step 10) and the failed transaction (step 12) were handled by the *TransactionScope* activity you placed in the workflow in step 5.

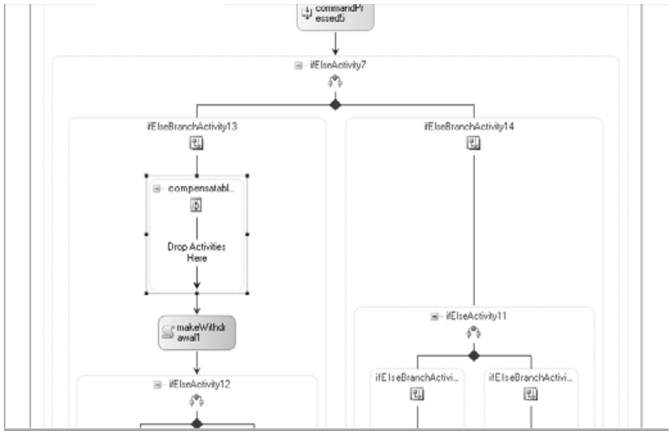
This is a phenomenal result! By including a single WF activity, we gained automatic (XA-style) transactional control over database updates. Can implementing a compensated transaction be as easy? As it happens, more work is required, but it's still not difficult to add compensated transactions to your workflow.

Adding a compensated transaction to your workflow

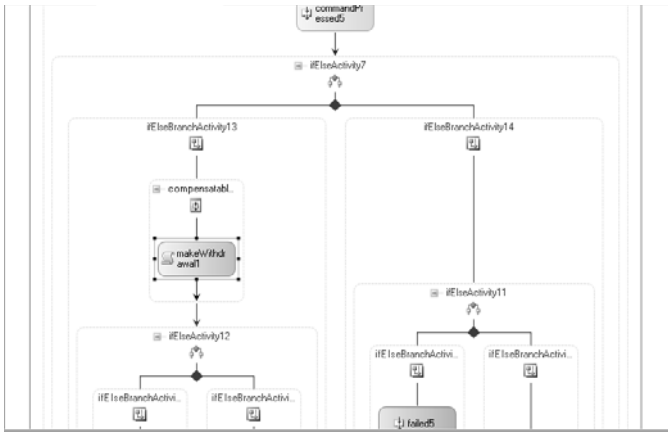
1. With the WorkflowATM solution open for editing, again open the Workflow1.cs file in the visual workflow designer. Look for the *WithdrawState* activity in the lower row of state activities and double-click the *CmdPressed5* activity you see there. This opens the *CmdPressed5* activity for editing, and once it's opened, you should see the *makeWithdrawal1 Code* activity on the left side of the workflow.



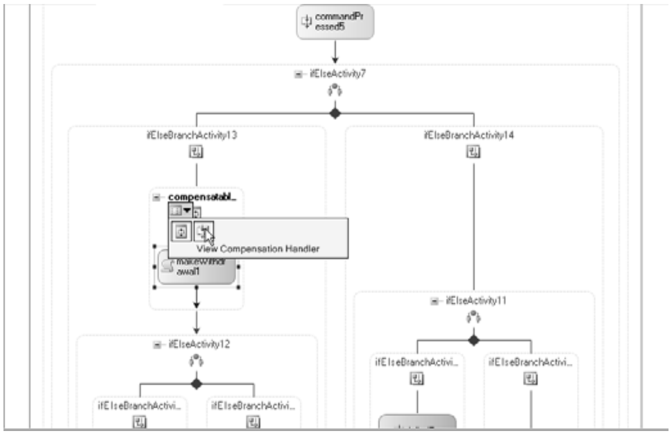
2. Similar to what you did with the preceding transaction, drag an instance of *CompensatableTransactionScope* from the Visual Studio Toolbox and drop it between the *makeWithdrawal1* activity and the *ifElseBranchActivity13* title above *makeWithdrawal1*.



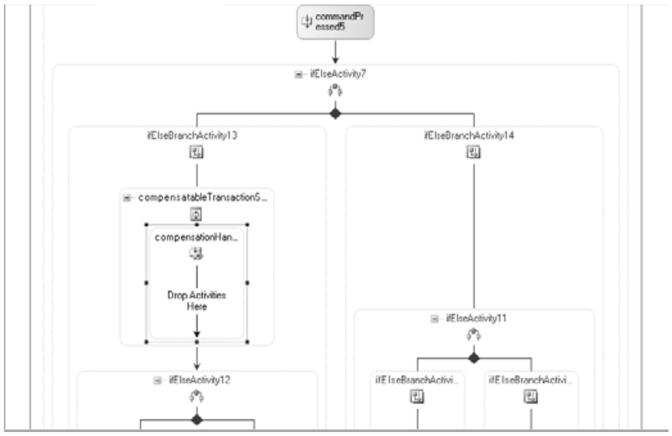
3. Drag the *makeWithdrawal1 Code* activity from below the *compensatableTransactionScope1* activity, and drop it into the transaction scope. The *MakeWithdrawal* method, which is bound to the *makeWithdrawal1* activity, now executes its ADO.NET code within an ambient transaction just as the deposit activity did.



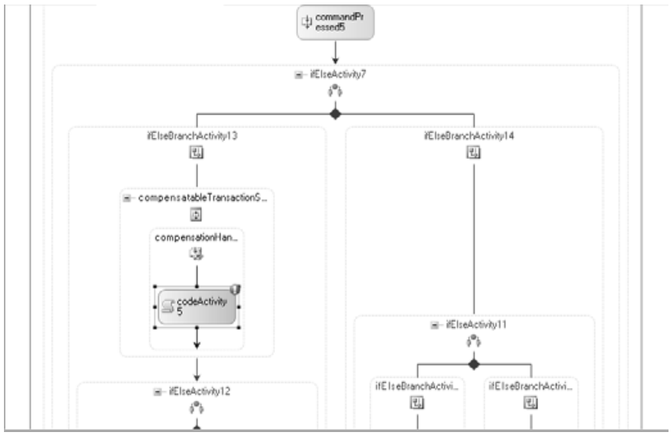
4. However, unlike the deposit functionality, you must provide the compensation logic. The transaction isn't rolled back in the traditional sense. Instead, you need to access the *compensatableTransactionScope1* compensation handler and add the compensating function yourself. To do that, move the mouse over the Smart Tag beneath the *compensatableTransactionScope1* title in the visual workflow designer and click it once to drop the view menu associated with this activity.



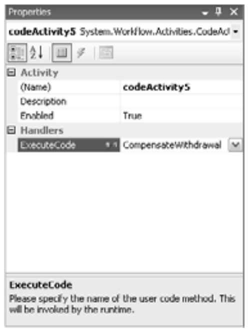
5. Click the right icon, View Compensation Handler, to activate the compensation handler view.



6. Drag an instance of the *Code* activity from the Visual Studio Toolbox, and drop it into the compensation handler activity.



7. For the *Code* activity's *ExecuteCode* property, enter **CompensateWithdrawal**. Visual Studio inserts the method into your source code and switches you to the code editor.


Properties

codeActivity5 System.Workflow.Activities.CodeAct

Activity	
(Name)	**codeActivity5**
Description	
Enabled	True
Handlers	
ExecuteCode	CompensateWithdrawal

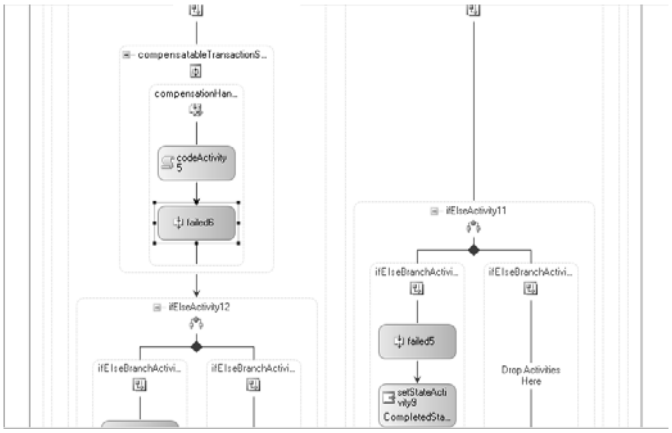
ExecuteCode
Please specify the name of the user code method. This will be invoked by the runtime.

8. Add the following code to the *CompensateWithdrawal* method you just inserted:

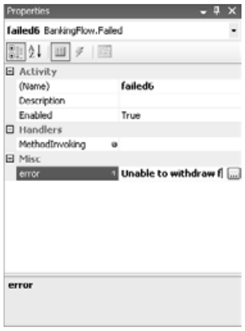
```
// Here, you "undo" whatever was done that did succeed. The
// code that withdrew the money from the account was actually
// successful (there is no catch block), so this compensation
// is forced. Therefore, we're safe in depositing the amount
// that was withdrawn. Note we can't use MakeDeposit since
// we require a SQL Server transaction and this method is
// called within the compensation handler (i.e., we can't drop
// a TransactionScope activity into the compensation to kick
// off the SQL Server transaction). We'll create the transaction
// ourselves here.
//
// Craft your compensation handlers carefully. Be sure you know
// what was successfully accomplished so that you can undo it
// correctly.
string connString =
     ConfigurationManager.ConnectionStrings["BankingDatabase"].
                                            ConnectionString;

if (!String.IsNullOrEmpty(connString))
{
    SqlConnection conn = null;
    SqlTransaction trans = null;
    try
    {
        // Create the connection
        conn = new SqlConnection(connString);

        // Create the command object
        SqlCommand cmd = new SqlCommand("dbo.Deposit", conn);
        cmd.CommandType = CommandType.StoredProcedure;

        // Create and add parameters
        SqlParameter parm = new SqlParameter("@AccountNo", SqlDbType.Int);
        parm.Direction = ParameterDirection.Input;
        parm.Value = _account;
        cmd.Parameters.Add(parm);
        parm = new SqlParameter("@ThrowError", SqlDbType.SmallInt);
        parm.Direction = ParameterDirection.Input;
        parm.Value = 0;
        cmd.Parameters.Add(parm);
        parm = new SqlParameter("@Amount", SqlDbType.Money);
        parm.Direction = ParameterDirection.Input;
        parm.Value = CurrentMoneyValue;
        cmd.Parameters.Add(parm);
        SqlParameter outParm =
                       new SqlParameter("@Balance", SqlDbType.Money);
        outParm.Direction = ParameterDirection.Output;
        outParm.Value = 0; // initialize to invalid
        cmd.Parameters.Add(outParm);

        // Open the connection
        conn.Open();
```

```
            // Initiate the SQL transaction
            trans = conn.BeginTransaction();
            cmd.Transaction = trans;

            // Execute the command
            cmd.ExecuteNonQuery();

            // Commit the SQL transaction
            trans.Commit();

            // Pull the output parameter and examine
            CurrentBalance = (decimal)outParm.Value;
        } // try
        catch
        {
            // Rollback... Note we could issue a workflow exception here
            // or continue trying to compensate (by writing a transactional
            // service). It would be wise to notify someone...
            if (trans != null) trans.Rollback();
        } // catch
        finally
        {
            // Close the connection
            if (conn != null) conn.Close();
        } // finally
    } // if
```

9. With the compensation code added to your workflow, return to the visual workflow designer and drop an instance of the custom *Failed* activity into the compensation handler, following the *Code* activity you just entered. Note Visual Studio might reformat and return you to the top-level state activity layout as you return to the visual workflow designer. If this happens, simply double-click the *CmdPressed5* activity in *WithdrawState* once again to access the *compensatableTransactionScope1* activity, and once again select the compensation handler view from its Smart Tag.



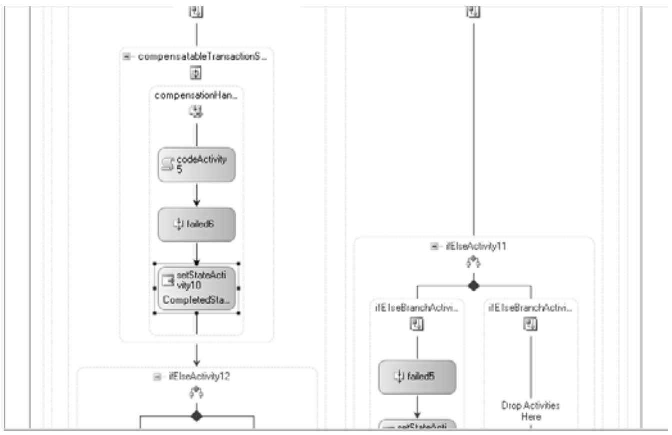
10. For the *Failed* activity's error property, type **Unable to withdraw funds**.


Properties
failed6 BankingFlow.Failed

Activity	
(Name)	failed6
Description	
Enabled	True
Handlers	
MethodInvoking	
Misc	
error	Unable to withdraw f

error

11. Following the *Failed* activity you just placed in your workflow, drag and drop an instance of *SetState*. For its *TargetStateName* property, select *CompletedState*.



12. To again test the application, press F5 or select Start Debugging from Visual Studio's Debug menu. After the application begins execution, click the B key to access the PIN verification screen, and then type **1234** (the PIN). Click the C key to verify the PIN and proceed to the activity selection screen.
13. Click the D key to make a withdrawal.
14. Press **10** to withdraw $100 (10 multiples of $10.00) and then click the D key to initiate the transaction. With no other intervention on your part, the transaction should succeed and the screen should now tell you the transaction is complete with a balance of $1234.56.
15. Now let's again force the transaction to fail. Click the C key to restart the ATM and then click the B key to access the PIN verification screen once more. Type **1234**, and then click the C key to access the banking activity selection screen.
16. Once again, enter **10** to withdraw another $100, and select the Force Transactional error check box. Then click the D key to initiate the transaction.

17. After you click the D key, the application indicates a transactional failure and displays the current balance ($1234.56). Because there is no *catch* block in the *MakeWithdrawal* method, we know the withdrawal was made. (If it was not, the application would have terminated with a critical error.) This means the account was in fact debited the $100 and that the compensating function ran, which added $100 back into the account.

Note There are other ways to see the account debited and then credited as well. You could set a breakpoint in the compensating function, or you could even execute SQL Server Profiler, if you're familiar with that application and are using the full retail version of SQL Server.

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 16, "Declarative Workflows." WF is capable of loading workflows declared in an XML format, and you'll see how that's accomplished in the next chapter.

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and close it. The next chapter introduces workflow in a slightly different way, and one you might find quite interesting, but there is certainly no hurry.

Chapter 15 Quick Reference

To	Do This
Introduce XA-style transactions into your workflow	Drop an instance of the *TransactionScope* activity into your workflow. You then should place all the transacted activities within the transactional scope. The ambient transaction will be applied to all, and should any one fail, all child activities will be rolled back. Otherwise, all will commit.
Introduce compensated transactions into your workflow	Drag and drop an instance of the *CompensatableTransactionScope* activity into your workflow. As with the *TransactionScope* activity, you then drop transacted child activities into the transactional scope. If all succeed, the transaction is considered successful and the child activities are committed. If not, the compensation handler is invoked and the code you place there to "undo" the transaction is executed.
Change the default order of compensation, or control which child transaction is compensated	Drop a *Compensate* activity into your compensation handler, cancellation handler, or fault handler. Assign the *TargetActivityName* property to the name of the activity to be compensated for.
Collect compensated transactions into a single work entity	Use the *CompensatableSequence* activity, and drop instances of the *CompensatableTransactionScope* activity into the compensated sequence. Keep in mind you can control which transactions are compensated using the *Compensate* activity. Individual compensated transactions do not require their own compensation function if the enclosing compensated sequence will provide for compensation.