BizTalk 2006 Recipes

A Problem-Solution Approach

Mark Beckner, Benjamin Goeltz,
Brandon Gross, Brennan O'Reilly,
Stephen Roger, Mark Smith, Alexander West

Apress®

BizTalk 2006 Recipes: A Problem-Solution Approach

ISBN-13 (pbk): 978-1-59059-711-8

ISBN-10 (pbk): 1-59059-711-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Stephen W. Thomas
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith, Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Lynn L'Heureux
Proofreader: Nancy Sixsmith
Indexer: Julie Grady
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Business Rules Framework

Business processes are changing at an unprecedented speed, and the IT systems that power business processes are being asked to keep up. In today's environment, in order to maintain a market advantage, organizations must react quickly to new markets, shifting consumer demand, shifting cost structures, and shifting business differentiators. The IT changes required to support dynamic business policies are taxing IT departments in terms of cycle time and hard costs.

At the root of this taxation are the complexities involved in designing and deploying traditional technical solutions. Current software development life cycle (SDLC) methodologies are time-consuming and expensive, and they often fail to deliver the set of features required by businesses within their time frames and budget targets. Although technologies such as .NET and BizTalk have shortened the cycle time to develop, test, and deploy solutions, often these technologies still don't meet the demanding cycle time requirements demanded by an agile business. For instance, consider a bank's business policy for granting credit to a customer. If you implement this logic within a BizTalk orchestration, the steps required to change the logic, compile the program, and redeploy are too numerous to enumerate here. In addition, this change requires that the system be taken offline while you enact the changes. Also, if the bank has a strong development methodology, it may require additional time to follow the process and obtain the necessary approvals. All this adds up to a sluggish, expensive process for changing the business policy.

The business rules framework is BizTalk's answer to reducing the business agility problem faced by traditional technologies. The business rules framework is a rules execution engine that allows you to create business policies that you can develop, deploy, and dynamically execute quickly without bringing your BizTalk process down. The framework comprises policies, facts, vocabularies, and rules. In conjunction, these artifacts combine to create a powerful tool for quickly changing business rules in near real time.

5-1. Creating a Business Policy

Problem

You want to understand the process for creating business rule policies. These policies are core components to all business rule development.

Solution

The business rules framework consists of policies, rules, vocabularies, and facts. A *business policy* is the container in which business rules are defined, versioned, tested, deployed, and executed. From within the business policy editing window, the first step before creating rules

is to create a policy. The policy then may contain one or more rules for deployment. To create a policy, take the following steps:

1. Open the Business Rule Composer by selecting Start ➤ Programs ➤ Microsoft BizTalk Server 2006 ➤ Business Rule Composer.
2. Within the Policy Explorer, right-click the Policies node, and click Add New Policy.
3. Give the policy a name.
4. By default, version 1.0 of the policy is created. You may change the version number if you want by selecting the version, navigating to the Properties window, and setting the `Version` property.
5. Add an optional description for the policy.
6. Right-click the version, and choose Save.

How It Works

Creating a business rules policy is a relatively simplistic exercise but is the foundation to the business rules framework. Once you have created the policy, you can add rules and facts to the policy for testing and deployment. When calling the policy from BizTalk, the latest version will be selected for execution, unless you specify a specific version. The rules engine detects and deploys new versions of a policy, allowing changes to be made to the business policy in a live, deployed process. This creates a powerful environment to apply a dynamic business policy to deployed solutions with minimal work. The recipes in this chapter will guide you through adding rules, facts, and vocabularies to your business rules policy.

5-2. Creating and Testing Rules

Problem

You need to create and test a simple business rule that a business process can use. The business process needs to validate a document and receive the result.

Solution

You need to perform a number of basic steps to create a rule. These include the following:

1. Define the rule logic.
2. Define the input schema and XML instance.
3. Define the output schema and XML instance.
4. Define the facts that the rule uses, such as constants, functions, and predicates (vocabularies).

The first step to creating a rule is determining the logic of the rule. This example will use the logic in Listing 5-1 to construct the rule.

Listing 5-1. *Rule Logic*

```
If Age < Minimum Age Then Deny Application
If Age > or = Minimum Age Then Accept Application
```

We will show how to execute the rule in Listing 5-1 against an XML document containing the data that will be validated. For this example, we will use the `NewHire` schema (Figure 5-1) with the schema and XML instance (Listing 5-2).

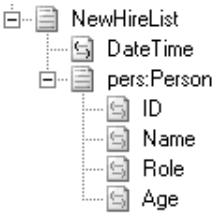

Figure 5-1. *NewHire schema*

Listing 5-2. *NewHire XML Instance*

```
<ns0:NewHireList xmlns:ns0="http://SampleSolution.NewHireList">
  <DateTime>1999-04-05T18:00:00</DateTime>
  <ns1:Person xmlns:ns1="http://SampleSolution.Person">
    <ID>1</ID>
    <Name>Mary</Name>
    <Role>Dog Walker</Role>
    <Age>17</Age>
  </ns1:Person>
  <ns1:Person xmlns:ns1="http://SampleSolution.Person">
    <ID>2</ID>
    <Name>Windy</Name>
    <Role>Baby Changer</Role>
    <Age>35</Age>
  </ns1:Person>
</ns0:NewHireList>
```

■ **Note** The schema must define the `<Age>` element as an integer for the comparison to work in the business rule, which you will create in this solution. The rules engine will not implicitly convert a String to an Int.

The next step in this solution is to output a document. The document that is sent to the rules engine will be modified and returned to the calling entity. If the applicant's age is not a valid age, the document will be modified to display the text "INVALID APPLICANT" in the `<Role>` element. (This element must exist in order for it to be set.)

The following steps describe how to create the vocabulary using the Business Rule Composer:

1. Open the Business Rule Composer from the Start menu. If the Open Rule Store dialog box appears, select the appropriate SQL Server instance and authentication to log on (this should be the instance against which you want to develop).

2. Add a new vocabulary, which will contain the definition for the `Minimum Age` constant and the `Age` node in the input XML document. In the Facts Explorer, right-click the `Constants` folder (or create a folder of your own—it could be named according to the rule, such as `AgeValidation`), and select Add New Version. Name it appropriately (the version number).

3. To add the `Minimum Age` constant, right-click the `Version` subfolder created in the previous step, and select Add New Definition. The Vocabulary Definition Wizard appears. Once the wizard is open, do the following:

   a. Add a new constant by selecting the Constant Value option. Enter **MinimumAge** for the Definition Name field. Give a description of the minimum age of the applicant. Click Next.

   b. On the final page of the wizard, set Type to System.Int32, and set Value to 18. Set Display Name to an appropriate value. Click Finish.

4. To add the reference to the `Age` node in the `NewHire` XML document, right-click the subfolder of the version that was created in step 2, and select Add New Definition. The wizard appears, and you should then perform the following steps (shown in Figure 5-2):

   a. Select the XML Document Element or Attribute option, and click Next.

   b. Enter **ApplicantAge** for the Definition Name field. Give a description of the actual age of the applicant.

   c. Click the Browse button, and find the `NewHire` XSD schema (based on Figure 5-1). Once you have selected the schema, a new dialog box appears where you can select the `Age` node. Select this node, and click OK.

   d. Modify the `Document type` namespace. This namespace must match the namespace of the schema that references it—ensure that it has been set correctly to `SampleSolution.NewHireList`.

   e. In the Select Operation section, select the Perform “Get” Operation radio button.

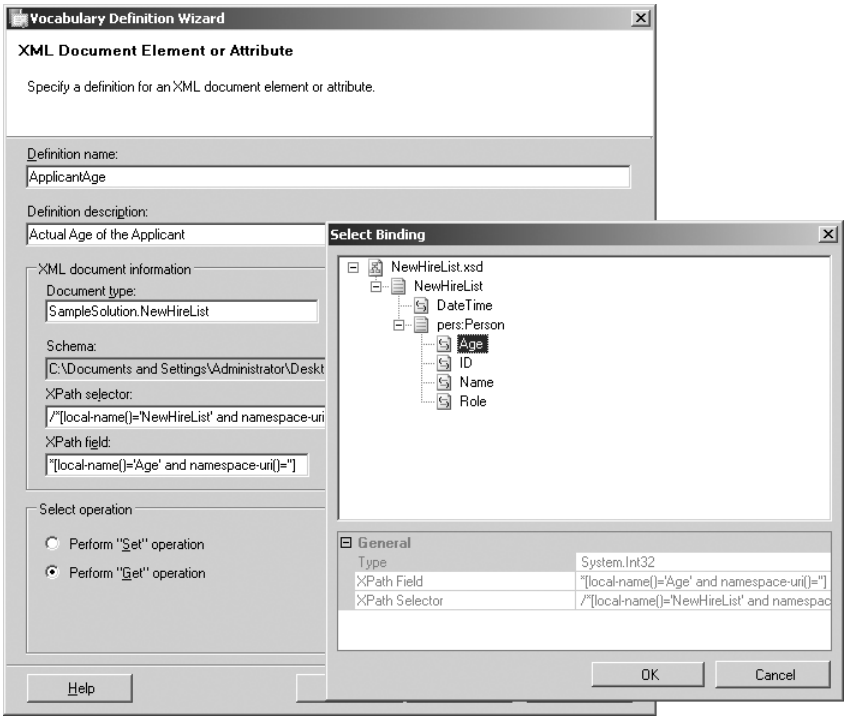

Figure 5-2. *Vocabulary Definition Wizard*

5. Add a reference to the `Role` node in the `NewHire` XML document. Following the same procedure as defined in step 4, use these substeps:

   a. Select the XML Document Element or Attribute option, and click Next.

   b. Enter **ApplicantRole** for the Definition Name field. Give a description of the role of the applicant, which will be set to INVALID APPLICANT.

   c. Click the Browse button, and find the `NewHire` XSD schema. Once you have selected the schema, a new dialog box appears where you can select the `Role` node. Select this node, and click OK.

   d. Modify the `Document type` namespace. This namespace must match the namespace of the schema that references it—ensure that it has been set to `SampleSolution.NewHireList`.

   e. In the Select Operation section, select the Perform "Set" Operation radio button.

6. Now publish the vocabulary by right-clicking the newly created `Vocabulary` folder created in step a and selecting Publish. This allows the vocabulary to be used by a policy, which is where you define an actual business rule.

The steps up to this point have been putting into place the components needed to create a business rule. A rule (or set of rules) is defined within a policy and references preexisting functions, predicates, and any custom-defined constants (such as what you defined in the previous steps). The following steps show how to create the actual business rule and policy and use the vocabulary created previously:

1. In the Policy Explorer, right-click the `Policies` folder, and select Add New Policy. Enter **SamplePolicy** for the policy name.

2. A version appears. Right-click this version, and select Add New Rule. Enter **SampleRule** for the rule name.

3. Select the rule you created in the previous step. You will see an empty condition in the right pane. Take the following steps to create the rule. (See Figure 5-3 for a full picture of the completed steps.)

   a. In the Facts Explorer, expand the `Predicates` folder, and select the `Less Than` predicate. Drag and drop this on the rule condition.

   b. On the `argument1` entry in the `IF` section of the Rule Composition window, drag and drop the custom vocabulary parameter `ApplicantAge` you created in an earlier step.

   c. On the `argument2` entry in the `IF` section, drag and drop the custom vocabulary parameter `MinimumAge` you created earlier.

   d. In the `THEN` section of the Rule Composition window, drop the `ApplicantRole` parameter (created earlier). Next to this you will see an `empty string` blank. Click this, and type **INVALID APPLICANT**.

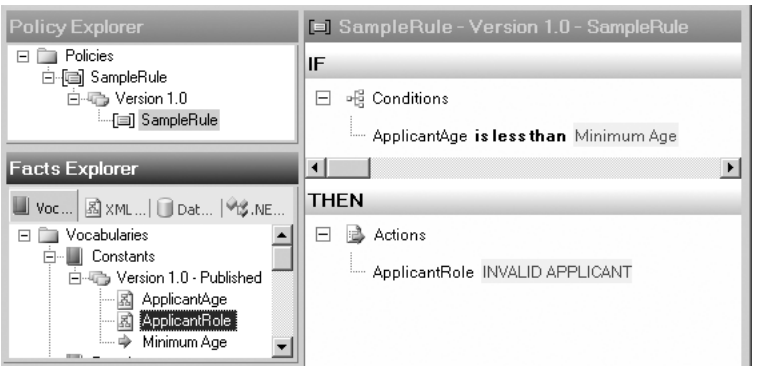

Figure 5-3. *Creating the rule*

The rule is complete at this point. Save the rule (right-click, and select Save). You can now test the rule (and policy). Use these steps to validate and test the rule:

1. Right-click the `Version` folder of the rule you just created, and select Test Policy.

2. In the Select Facts dialog box that appears, click the schema name. The option Add an Instance of the NewHireList XML will be available. You saw an example of this earlier in this recipe (refer to Listing 5-2). Click Add Instance, and browse to an instance of this XML document.

3. Once you have declared an instance, click the Test button. This causes the XML instance to be passed to the rule. A full trace appears in an Output window where you can verify whether the rule executed as expected (the value of the XML node should change).
4. Right-click the policy, and select Publish.
5. Right-click the policy, and select Deploy.

You can now call the rule and policy from an orchestration. To call the rule and policy from an orchestration, perform the following steps:

1. In an orchestration, create a message of the type that is to be passed into the rule and a message of the type that is expected back (in this case, the messages will both be the same `NewHireList` XSD type).
2. Drop a Call Rules shape in an Atomic Scope (or in an orchestration that has a transaction type of `Atomic`).
3. Right-click the Call Rules shape, and select the policy you created. You will see one input parameter, which will be the message created in step 1.

■ **Note** The document type must match the namespace of the schema that will call it for the rule to be callable from an orchestration.

The XML instance shown in Listing 5-3 represents the generated output from testing the business rule created in this solution.

Listing 5-3. *Output XML Instance*

```
<ns0:NewHireList xmlns:ns0="http://SampleSolution.NewHireList">
  <DateTime>1999-04-05T18:00:00</DateTime>
  <ns1:Person xmlns:ns1="http://SampleSolution.Person">
    <ID>1</ID>
    <Name>Mary</Name>
    <Role>INVALID APPLICANT</Role>
    <Age>17</Age>
  </ns1:Person>
  <ns1:Person xmlns:ns1="http://SampleSolution.Person">
    <ID>2</ID>
    <Name>Windy</Name>
    <Role>Baby Changer</Role>
    <Age>35</Age>
  </ns1:Person>
</ns0:NewHireList>
```

How It Works

There are additional uses of the rules engine, including the ability to store parameters that can be accessed from orchestrations. Similar in nature to a configuration file that stores parameters that may need to change (such as connection strings, etc.) once a solution is in production, these constants allow for runtime modifications. Benefits to using the rules engine to store configurable parameters include ease of access, simple user interface, and enhanced read/write permissions (compared with a text file containing configurable parameters).

To add parameters that can be accessed from an orchestration, there are two basic steps: create the parameters, and access the constants through the rules engine API.

The first step is to create the parameters in a vocabulary in the rules engine and save them. You don't need to publish them unless versioning would be helpful. If they are not published, it is easier to modify them at runtime, because all that is required is to right-click the value that needs to be changed and select Modify. Once you have saved the vocabulary, running instances of orchestrations that reference these parameters will pick up the new values.

The second step is to add .NET code to a BizTalk `Helper` class to access the vocabulary through the rules engine API. The code shown in Listing 5-4 can be placed into a .NET assembly and should be referenced by a BizTalk project. The method can then be called from an Expression shape within an orchestration. This code also allows for a specific version to be called, if desired.

Listing 5-4. *Accessing Constants Through the Rules Engine API*

```
public bool blnGetVocabularyConstantValue(string strVocabularyName,
                                    string strConstantName,
                                    string strVersion,
                                    ref string strConstantValue,
                                    ref string strConstantType) {

   RuleStore rlsRuleStore;
   VocabularyInfoCollection vicVocabInfo;
   Vocabulary vocVocab;
   RuleSetDeploymentDriver rsdDriver = new RuleSetDeploymentDriver();
   Hashtable hshTable;
   LiteralDefinition litDef;
   Boolean blnConstantFound = false;

   rlsRuleStore = rsdDriver.GetRuleStore();

   if((strVersion == null) || (strVersion == "")) {
    vicVocabInfo = rlsRuleStore.GetVocabularies(strVocabularyName,
                              RuleStore.Filter.Latest);
   }
   else {
    vicVocabInfo = rlsRuleStore.GetVocabularies(strVocabularyName,
                              RuleStore.Filter.All);
   }
```

```
   // check to see that the vocabulary searched on has returned data
   if(vicVocabInfo.Count > 0) {
    for(int x=0;x<vicVocabInfo.Count;x++) {

    vocVocab = rlsRuleStore.GetVocabulary(vicVocabInfo[x]);

    // Check to see that the version equals the version being searched for
    // or that the version passed in was a wildcard (Null)
    if((strVersion == null) || (strVersion == "") ||
    ((vocVocab.CurrentVersion.MajorRevision + "." +
      vocVocab.CurrentVersion.MinorRevision) == strVersion)) {

     // Initialize the hashtable to the number of definitions on the vocabulary
     hshTable = new Hashtable(vocVocab.Definitions.Count);
     hshTable = (Hashtable)vocVocab.Definitions.SyncRoot;

     // check that the constant being searched for exists in the vocabulary
     if(hshTable.ContainsKey(strConstantName)) {
      litDef = (LiteralDefinition)hshTable[strConstantName];

      strConstantType = litDef.Value.GetType().ToString();
      strConstantValue = litDef.Value.ToString();

      blnConstantFound = true;

      // exit loop
      break;
     }
    }
   }
  }
 return blnConstantFound;
}
```

You can use the code in Listing 5-5 to invoke the previous class from an orchestration Expression shape. This allows for an alternate method of access to the rules engine without needing the standard call using the Call Rules shape and is useful when getting the value of a constant or other scalar value.

Listing 5-5. *Calling the Object from the Orchestration Expression Shape*

```
// the following string is the name of the vocabulary
strVocabularyName = "Project.BizTalk.Constants";
objVocab.GetVocabularyConstantValue(strVocabularyName, "ConstantName", "",
                                                    "10", ref strValue,ref strType);
```

5-3. Creating Facts

Problem

You need to understand how to use the Facts Explorer to create a vocabulary that will be used within a business rule fact. You want to be able to store constants, predicates, and so on, that can be changed easily without redeploying code.

Solution

This solution will demonstrate how to create a vocabulary using a node in an imported XML schema. *Facts* are those items that are used to create rules. The Facts Explorer has four tabs, as follows:

- *Vocabularies*: These consist of all defined values that you can use when creating a rule, including constants, predicates, XML nodes, and so on.
- *XML Schemas*: You can use all schemas imported onto this tab when creating vocabularies or predicates (actions). You can drag and drop nodes on the XML Schemas tab in the Vocabulary window.
- *Databases*: You can add references to databases that will be used for creating facts on this tab. You can drag and drop tables onto the Vocabulary window.
- *.NET Assemblies*: Assemblies, like databases and XML schemas, can be references in the Facts Explorer and used to create vocabularies.

Use the following steps to import an XML schema and create a new vocabulary:

1. In the BizTalk Business Rule Composer, click the XML Schemas tab in the Facts Explorer.
2. Right-click the `Schemas` folder, and select Browse. Locate a schema to import, and click Open. This imports the full schema into the window, as shown in Figure 5-4.

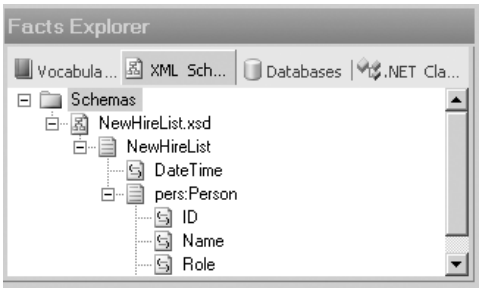

Figure 5-4. *XML Schemas tab*

3. Create a new vocabulary by clicking the Vocabulary tab. Right-click the `Vocabularies` folder, and select Add New Vocabulary. Give the vocabulary a name.

4. Now drag a node from the XML schema that was imported, and drop it on the vocabulary you created in the previous step. By holding down the mouse button as you drag the node, you can navigate between tabs.

5. The Vocabulary Definition Wizard immediately opens, with the XML Document or Attribute option automatically selected. Navigate through the wizard to finish creating the vocabulary. The fields will already be filled in for you.

How It Works

You can add the Database and .NET Assembly facts to the Vocabulary tab in a similar manner as described in this solution. Additionally, you can drop all facts in the condition or action (`IF...THEN`) of a rule. Once you have added a fact to a rule, right-clicking the condition or action allows you to browse to the original fact, as shown in Figure 5-5. Complex rules will have many facts referenced in them, and this allows you to keep track of all the disparate definitions.

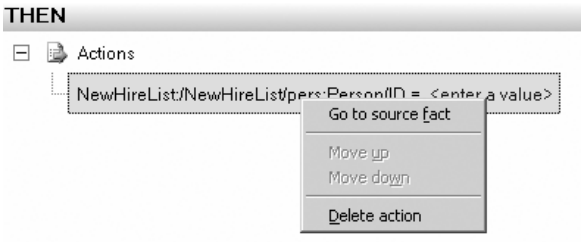

Figure 5-5. *Action with fact*

5-4. Setting Rule Priorities

Problem

You need to control the order in which the BizTalk rules engine executes rules. The order BizTalk executes rules in determines which rules take precedence, with the results of the last rule executing and potentially overwriting the results of the previous rules.

Solution

In many real-world scenarios, multiple business rules must execute in a predetermined order. In these situations, the second rule may overwrite the result of the first rule. Control the order in which the BizTalk rules engine executes rules by assigning a priority to each rule.

This recipe defines two business rules for determining which shipper should deliver a package to a customer. A customer can specify a preferred shipper to deliver their items. However, sometimes customers receive promotional discounted shipping rates, and a customer can specify they would like to take advantage of these special shipping rates. The promotional shipping rate rule must execute after and override the results of the preferred shipper rule.

■ **Note** This example assumes that the business rules already exist. For more information about creating business rules, please see the earlier recipes in this chapter.

The following steps demonstrate how to set the priority of a business rule:

1. From the Start menu, select All Programs ➤ Microsoft BizTalk Server 2006 ➤ Business Rule Composer.
2. Navigate to the policy containing the shipping business rules.
3. Select the Use Preferred Shipper rule in the Policy Explorer, and change the `Priority` property to the value 1, as shown in Figure 5-6.

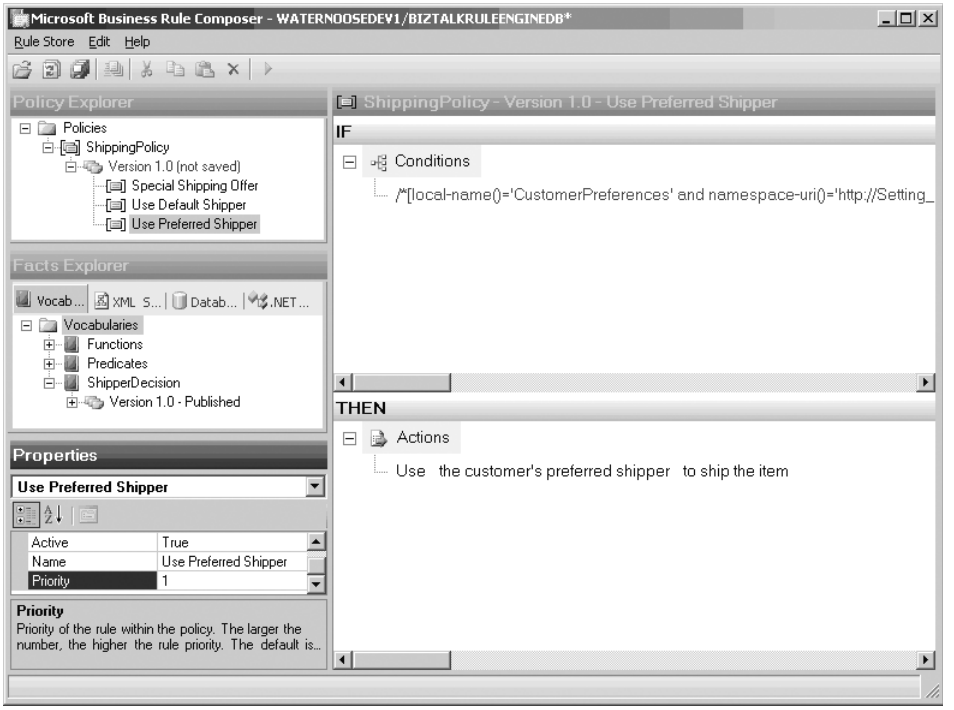

Figure 5-6. *Setting the Use Preferred Shipper rule's priority*

■ **Note** The BizTalk rules engine executes higher-priority rules first. Since the promotional shipping rate rule should override the preferred shipper rule, it must execute after the preferred shipper rule and must have a lower priority.

4. Select the Special Shipping Offer rule in the Policy Explorer, and verify that the Priority value is set to the value 0.

How It Works

The BizTalk rules engine executes high-priority rules first. Consequently, lower-priority rules execute after higher-priority ones and potentially can override the results of higher-priority rules. The BizTalk rules engine may execute rules with the same priority in any order. Remember that the priority of rules will affect the order BizTalk executes them in and has no direct effect on the importance of the rule's actions.

5-5. Creating Custom Fact Retrievers

Problem

You need to define business rules using facts that an orchestration cannot provide. The facts may not be available in the orchestration, or you may need to reuse a fact across many instances of the orchestration.

Solution

A business rule performs actions when conditions defined in the Business Rule Composer are true. For example, an action may modify a value in a message. The BizTalk developer creates a condition from facts, which are pieces of information the rules engine can examine. Most often, the rules engine will examine facts from the messages BizTalk directly processes. The orchestration invoking the rules engine directly provides these facts. However, sometimes a rule needs to be based on facts that are not available in the orchestration. The rules engine can retrieve these external facts with a custom fact retriever.

In this example, a business rule will schedule a customer's service request only if the customer's address is in the Northwind database. The customer's address may be different each time the orchestration invokes the rules engine. The customer's address is called a *short-term* fact because the orchestration provides the fact every time it invokes the rules engine. However, instead of opening a new connection to the database each time the orchestration invokes the rules engine, this example reuses the same connection to the Northwind database over and over again. Reusing the database connection makes it a *long-term* fact. The BizTalk developer must provide long-term facts to the rules engine with the `IFactRetriever` interface. The custom fact retriever created in this recipe provides the connection to the Northwind database to the rules engine.

Use the following steps to create the custom fact retriever:

1. Open Visual Studio .NET, and create a new class library project.
2. Add a reference to the Microsoft.RuleEngine.dll assembly. This assembly is located in the root of the BizTalk installation directory. The default location of this folder is `C:\Program Files\Microsoft BizTalk Server 2006\`.
3. Implement the `Microsoft.RuleEngine.IFactRetriever` interface. This interface defines one method called `UpdateFacts`.
4. Within the `UpdateFacts` method, add code to check whether the `factsHandleIn` parameter is `null`. If the `factsHandleIn` parameter is `null`, create and return an instance of the `Microsoft.RuleEngine.DataConnection` class. If the `factsHandleIn` parameter is not `null`, then simply return it from the method. When completed, the code should appear as in Listing 5-6.

Listing 5-6. *Completed Custom Fact Retriever*

```
using System.Data.SqlClient;
using Microsoft.RuleEngine;

namespace CustomFactRetriever
{
    public class AssertDBConnection : IFactRetriever
    {
        public object UpdateFacts(RuleSetInfo ruleSetInfo,
                                  RuleEngine engine, object factsHandleIn)
        {
            object factsHandleOut;
            if (factsHandleIn == null)
            {
                SqlConnection SQLConn = new SqlConnection("
                                                Initial Catalog=Northwind;
                                                Data Source=(local);
                                                Integrated Security=SSPI;");
                DataConnection RulesConn = new DataConnection("Northwind",
                                                    "Customers", SQLConn);
                engine.Assert(RulesConn);
                factsHandleOut = RulesConn;
            }
            else
                factsHandleOut = factsHandleIn;

            return factsHandleOut;
        }
    }
}
```

5. Compile the class library assembly, and deploy it to the Global Assembly Cache (GAC). After creating the custom fact retriever, define a vocabulary and business rules. This example will schedule a service appointment only if the customer's address is in the database.

6. Open the Business Rule Composer, and create a new policy with one rule. Define a vocabulary for accessing facts defined by an XML message. An orchestration must provide the message when the policy executes. This example gets the address a customer requested service at and decides whether to approve the service request.

7. Right-click the new vocabulary, and select Add New Definition. Select the option to create a new database table or column definition in the vocabulary, and click Next, as shown in Figure 5-7.

8. Give the definition an appropriate name, and leave the Binding Type set to Data Connection.

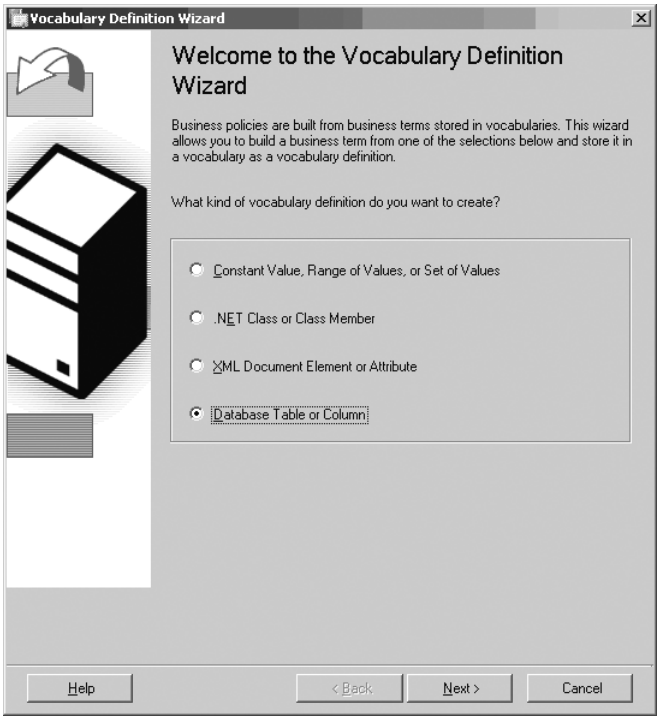

Figure 5-7. *Creating a new database column definition*

9. Within the Database Information section of the window, click the Browse button. After selecting a SQL Server instance with a Northwind database, browse to the Customer table. Select the Address column of the Customer table, as shown in Figure 5-8, and click OK.

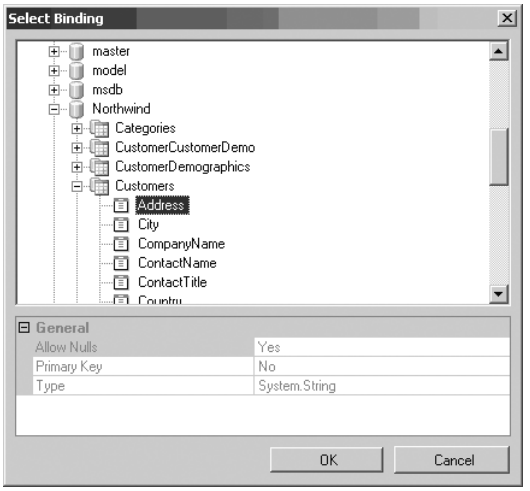

Figure 5-8. *Setting the location of the customer's registered address*

10. Within the Select Operation section of the window, select the Perform "Get" Operation radio button. Set an appropriate display name for display in the Business Rules Composer. When complete, the Database Table or Database Table Column Definition window appears, as shown in Figure 5-9.

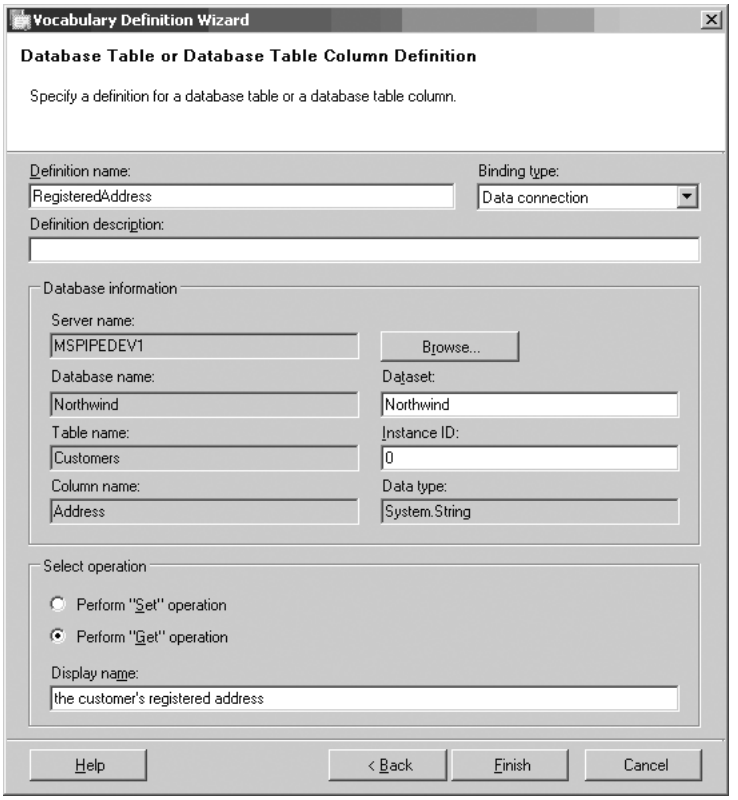

Figure 5-9. *Completing the definition*

11. Click the Finish button to complete the vocabulary definition.

12. After creating the vocabulary definition, specify the custom fact retriever to provide the connection to the database when the rule executes. Then, select the policy version in the Policy Explorer section of the Business Rule Composer. The properties should appear in the lower-left region of the Business Rule Composer.

13. Select the `Fact Retriever` property of the policy version, and click the ellipsis that appears.

14. In the Select Configuration component, click the Browse button. In the list of assemblies that appears, select the custom fact retriever created previously in this example.

■ **Note** If you cannot locate the custom fact retriever, verify that it is deployed to the GAC.

---

15. Select the class implementing `UpdateFacts` to complete configuring the custom fact retriever, as shown in Figure 5-10.

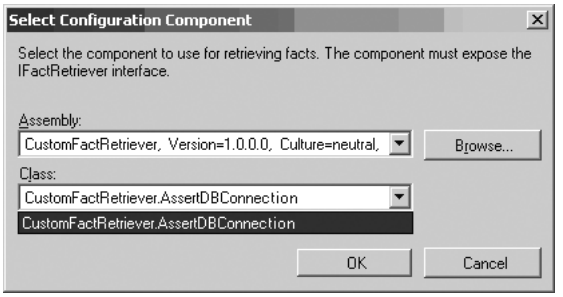

Figure 5-10. *Configuring the custom fact checker*

16. After creating the custom fact retriever to retrieve the `DataConnection` and configuring the policy to use the custom fact retriever, it is time to define the rules. So, define the business rule to compare the requested service address with the customer addresses already in the Northwind database. If there is a match, then the service request will be scheduled. When completed, the rule should appear, as shown in Figure 5-11.

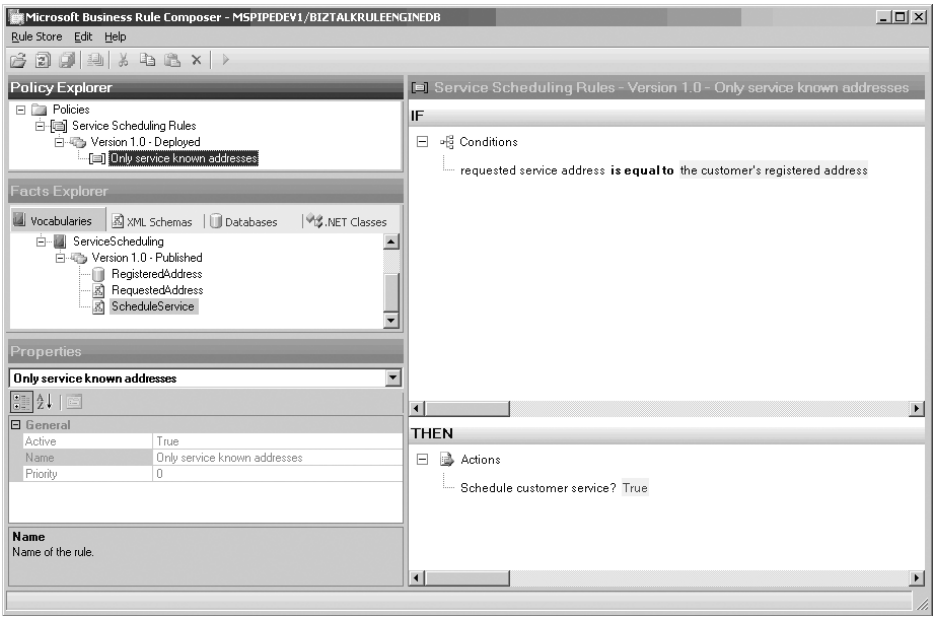

Figure 5-11. *Completed service address verification rule*

17. Deploy the rules and invoke them from an orchestration, specifying the XML messages that the policy expects. The rules engine will invoke the `UpdateFacts` method of the custom fact retriever. The method will add a `DataConnection` fact to the Northwind database to the rules engine facts.

How It Works

The rules engine examines facts provided either directly from an orchestration or from a custom fact retriever. The facts provided directly from an orchestration are called *short-term* facts, and the orchestration supplies them each time BizTalk invokes the rules engine. The facts provided from a custom fact retriever are called *long-term* facts, and they can be reused each time BizTalk invokes the rules engine.

A BizTalk orchestration can also treat the `DataConnection` in this example as a short-term fact. Accomplish this by creating a `DataConnection` in an Expression shape and including it as a parameter to the Call Rules shape. However, each orchestration instance would create its own `DataConnection`. You can improve the scalability and performance of the application by treating the `DataConnection` as a long-term fact and sharing it each time BizTalk invokes the rules engine.

In general, treating a fact as long-term is preferable under the following conditions:

- The fact value can be shared each time BizTalk invokes the rules engine.
- Retrieving the fact incurs a significant performance penalty.
- Caching the fact value will still allow the rules to execute correctly.
- The fact is not directly available in the orchestration that invokes the rules.

The rules engine invokes the custom fact retriever's `UpdateFacts` method each time the policy executes. This method decides when to update the facts. In this example, the `UpdateFacts` method examines the `factsHandleIn` parameter to check whether the `DataConnection` was already created. If the `factsHandleIn` parameter is `null`, then the method creates the `DataConnection`, inserts it into the rules engine, and completes by returning the `DataConnection` object. The next time the rules engine invokes the method, the rules engine includes the object returned by the method in the `factsHandleIn` parameter. If the `factsHandleIn` parameter is not `null`, the method knows that it has already inserted the `DataConnection` into the rules engine, and it simply returns the same object.

Although this solution example uses a simple algorithm to decide when the long-term fact needs updating, the developer can implement a more sophisticated approach. For example, the developer could refresh the facts every hour by returning a `DateTime` object when the facts are refreshed and decide to update facts by comparing the last refresh time to the current time.

5-6. Calling the Business Rules Engine from .NET

Problem

You are building a solution that must execute business rules based on dynamic information. These business rules are likely to change over time, and you want to minimize the impact on the solution when modifications are needed. To do this, you want to be able to call the rules engine from a .NET assembly.

■ **Note** The problem within this recipe is the same as in Recipe 5-7. The solutions in the two recipes differ in how the business rules are executed.

Solution

This solution describes the steps necessary to call the business rules engine from a C# assembly. The steps detail how to build a .NET Windows application to process job applications and dynamically determine whether applicants meet a minimum age requirement. If the applicant is younger than 18, then they cannot be considered for employment. The minimum age is likely to change over time and needs to be easily configurable. The business rules engine that comes with BizTalk Server captures the business rule determining the minimum working age. Other recipes within this chapter detail the steps for creating rules.

■ **Note** This recipe is based on the policy and vocabulary defined in Recipe 5-2.

This solution assumes that a rule exists in the business rules engine to determine whether an applicant meets the minimum age requirements. To call this rule from a .NET application, follow these steps:

1. Open the project containing the .NET application.
2. Add a project reference to the `<Install Folder>\Microsoft BizTalk Server 2006\Microsoft.RuleEngine.dll` assembly, which contains the classes required to call the business rules engine.
3. Create the necessary policy and fact objects, and execute the business rule policy, as shown in Listing 5-7.

Listing 5-7. *Calling a Business Rule from .NET*

```
public void callSamplePolicy(ref System.Xml.XmlDocument newHireListDoc)
{
    // create the SamplePolicy policy object
    // specify policy name and version
    Microsoft.RuleEngine.Policy policy =
             new Microsoft.RuleEngine.Policy("SamplePolicy", 1, 0);

    // create the facts object array to hold the input parameters for the policy
    object[] facts = new object[1];

    // create the input parameter for the SamplePolicy policy
    // based on a typed BizTalk schema (fully qualified .NET type)
    Microsoft.RuleEngine.TypedXmlDocument typedXmlDoc =
      new Microsoft.RuleEngine.TypedXmlDocument("SampleSolution.NewHireList",
                                                              newHireListDoc);

    // add the input parameter to the facts object array
    facts[0] = typedXmlDoc;
```

```
    // execute the policy against the facts
    policy.Execute(facts);
    policy.Dispose();

    // set the parameter object
    newHireListDoc.LoadXml(typedXmlDoc.Document.OuterXml);
}
```

How It Works

Although the business rules engine comes as part of BizTalk Server, this solution shows that .NET assemblies outside the BizTalk environment can call into it. This allows external applications to use the same rule framework that the integration hub does, enabling companies to consolidate their business rules functionality onto one platform.

■ **Note** You can extend this solution by using a web service method to access the business rules engine, allowing code on any platform to call into a common rule framework.

For a .NET project to call into the business rules engine, it must reference the `Microsoft.RuleEngine.dll` assembly. This assembly contains the classes used to access the rules framework, including those to execute policies. In this solution, you first create an instance of the policy object, with the appropriate name and version. The name must exactly match the name of the policy you want to execute, as shown in Figure 5-12. This solution specifies `SamplePolicy` as the policy name, which maps to the highlighted name in Figure 5-12.

Following the policy name, you specify the major and minor versions of the policy you want to execute. You specify 1.0, which is the only version of the policy currently deployed. Alternatively, you could specify the name only when creating the policy object (no version parameters supplied to the policy's constructor method), which executes the most recently deployed version of the policy.

Next, you create a collection (array) of fact objects. This array holds all the objects necessary to execute the policy. These objects map to the different types of facts that the Business Rule Composer can create, including XML documents, .NET classes or class members, and database connections. The `SamplePolicy` policy uses only a single XML document fact, which you create next.

You use the `Microsoft.RuleEngine.TypedXmlDocument` class to create an XML document fact, specifying the document type (the fully qualified .NET type name) and XML document instance. Add this `TypedXmlDocument` to the fact collection.

Finally, you execute the policy against the facts collection. As shown in Recipe 5-7, the facts collection is passed as a *reference* parameter, meaning any changes to the facts will be committed to the original instances. In this solution, the XML document fact will have the appropriate minimum age requirement logic applied.

It is also possible to execute a policy directly from an Expression shape within an orchestration. Listing 5-8 illustrates how you would execute the same policy as described in the "Solution" section of this recipe but from an Expression shape.

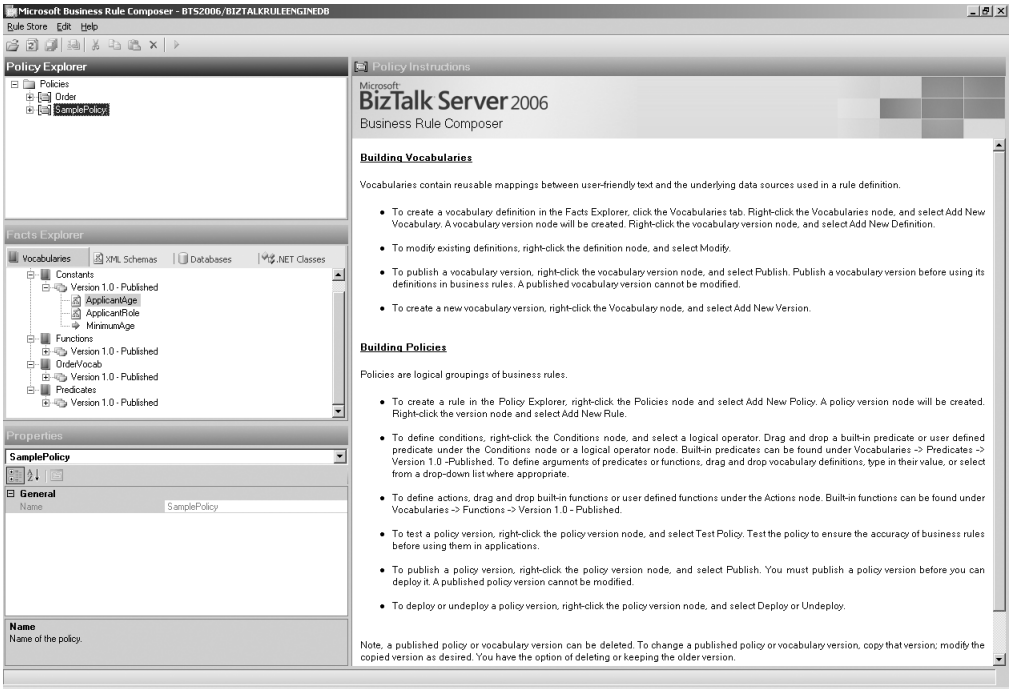

Figure 5-12. *Specifying a policy name*

Listing 5-8. *Calling Policy from Expression Shape*

```
// create the SamplePolicy policy object
// policy variable has type of Microsoft.RuleEngine.Policy
policy = new Microsoft.RuleEngine.Policy("SamplePolicy", 1, 0);

// create the input parameter for the OrderShipping policy
// typedXmlDoc variable has type of Microsoft.RuleEngine.TypedXmlDocument
// NewHireListMessage has type of SampleSolution.NewHireList
typedXmlDoc =
  new Microsoft.RuleEngine.TypedXmlDocument("SampleSolution.NewHireList",
                                                    NewHireListMessage);

// execute the policy against the facts
policy.Execute(typedXmlDoc);
policy.Dispose();
```

Although the policy executed in this recipe requires only a single XML document fact, facts can also be .NET classes or class members and data connections. By default, BizTalk Server requires you to provide an object instance for each fact used in a policy.

Data Connection Facts

To pass a data connection fact to a policy, initialize a `Microsoft.RuleEngine.DataConnection` instance and pass it to the policy in the facts collection. If an update is being executed against the underlying database, you must also provide a transaction to the `DataConnection` object, as shown in Listing 5-9. If data is being retrieved only, no transaction is required.

Listing 5-9. *Providing a Transaction*

```
// create SQL connection object
sqlConnection = new System.Data.SqlClient.SqlConnection("ConnectionString");

// create SQL transaction object
sqlTransaction = sqlConnection.BeginTransaction();

// create Data Connection object
dataConnection = new Microsoft.RuleEngine.DataConnection(
        "SqlDataSetName",
        "SqlTableName",
        sqlConnection,
        sqlTransaction);
```

The .NET assembly can add this `DataConnection` object instance to a fact collection and pass it to the policy for execution.

5-7. Calling the Business Rules Engine from an Orchestration

Problem

You are building a workflow that must execute business rules based on dynamic information. These business rules are likely to change over time, and you want to minimize the impact on the solution when modifications are needed.

■ **Note** The problem within this recipe is the same as in Recipe 5-6. The solutions in the two recipes differ in how the business rules are executed.

Solution

This solution will demonstrate how to call the business rules engine from an orchestration, without using a .NET assembly. The basic problem is similar to the previous recipe (Recipe 5-6). The key points of the problem are as follows:

- Job applications must be processed, and it must be determined whether they meet the minimum age requirement.
- If the applicant is younger than 18, then they cannot be considered for employment.
- The minimum age is likely to change over time and needs to be easily configurable.

Note This recipe is based on the policy and vocabulary defined in Recipe 5-2.

This solution assumes that a rule exists in the business rules engine to determine whether an applicant meets the minimum age requirements. To call this rule from an orchestration, follow these steps:

1. Open the project containing the orchestration.
2. In the orchestration's Properties window, configure it to act as a long-running transaction.
3. Create a new message, and specify the name and type. In this scenario, create a message named `NewHireListMessage` defined by the `NewHireList` schema.
4. From the Toolbox, drag the following onto the design surface in top-down order:
   a. Receive shape to receive the initial order message—configure this shape to use the `NewHireListMessage` message, to activate the orchestration instance, and to use an orchestration receive port.
   b. Send the shape to deliver the `NewHireListMessage` message to an external system—configure this shape to use an orchestration send port.
5. From the Toolbox, drag a Scope shape onto the design surface, in between the Receive and Send shapes configured in the previous step—configure this shape to act as an Atomic transaction, as shown in Figure 5-13.

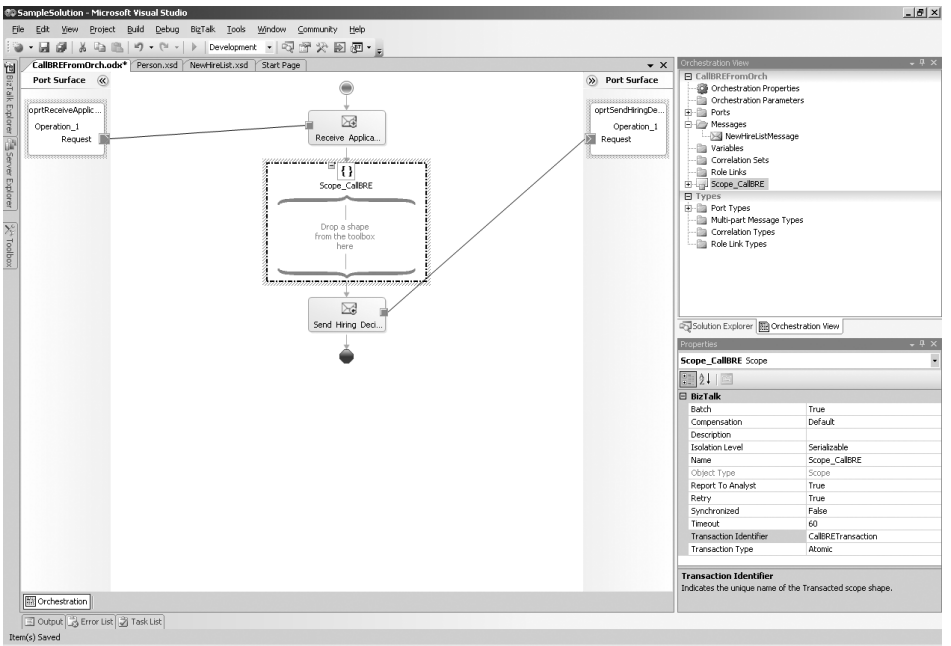

Figure 5-13. *Configuring the Scope shape*

6. From the Toolbox, drag a Call Rules shape onto the design surface, inside the Scope shape configured in the previous step. Configure this shape by double-clicking it, which launches the CallRules Policy Configuration dialog box, as shown in Figure 5-14. Select the appropriate business policy to call and the parameters to pass into the policy (`SamplePolicy` and `NewHireListMessage` in this scenario).

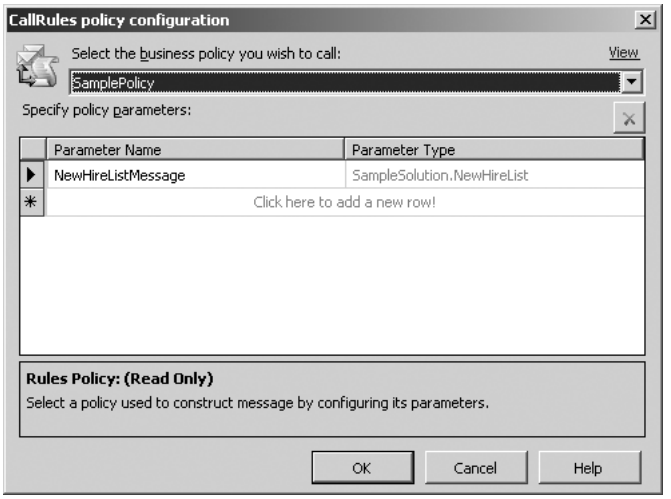

Figure 5-14. *Configuring the Call Rules shape*

■ **Note** The Parameter Type list populates with all the data types expected as input parameters to the selected business policy. The Parameter Name list populates with all the orchestration variables and messages matching the parameter types.

How It Works

Calling the business rules engine from an orchestration is a straightforward task. It allows you to separate the business procedure and process flow from the business rules. This is particularly useful when the rules are used across many process flows, change frequently, or need audit logging and versioning capabilities.

This recipe's solution uses a message instance conforming to the `NewHireList` schema as an input parameter to the `SamplePolicy` business rule policy. This policy sets the `<Role>` element of the message to INVALID APPLICANT if the `<Age>` element is less than 18. You could then enhance the rule to apply different minimum age requirements for different roles or to retrieve real-time minimum age requirements from a government-run data source.

You could easily extend this solution to illustrate how the output from a business rule can facilitate business procedures and process flow within an orchestration. Business rule policies could define a complex set of rules to determine whether an applicant is valid for a role. As opposed to setting the `<Role>` element to INVALID APPLICANT, you could configure the policy to return a Boolean value indicating whether the applicant meets the minimum

age requirements. The orchestration could then use this Boolean value in a Decide shape, allowing separate process flows to determine whether an applicant is valid for a role based on previous work experience, personal references, and salary requirements.

For an orchestration to successfully call a policy, you must make a few settings:

- The Call Rules shape must be within an Atomic Scope shape (this necessitates that the orchestration's `Transaction` property is set to Long Running).
- The Policy must be published and deployed within the business rules engine.

■ **Note** The Call Rules shape will always access the most recently deployed version of a policy.

- An initialized instance of all variables referenced in the policy must be in the same scope as the Call Rules shape.

Although this solution uses only an XML message instance as input to the rules policy, policies can accept other types of input parameters as well. The Vocabulary Definition Wizard allows a .NET class/class member or a data table/column to be used as business terms. As previously stated, an initialized instance of all variables referenced in a policy must be in scope at the Call Rules shape in the orchestration. This means that if a .NET class and a data table/column are used in a policy, there must be an initialized instance of the .NET class and rules engine `Microsoft.RuleEngine.DataConnection` object available in the orchestration to pass into the policy as input parameters.

■ **Note** If there is no instance matching an input used in a policy's rules, that input parameter will not appear in the Call Rules Policy Configuration dialog box's Parameter list. This will prevent the policy from being successfully called from the orchestration.

The policy used in this scenario references vocabulary definitions based on two XML elements in the `NewHireList` schema: `<Age>` and `<Role>`. Take care when creating these vocabulary definitions, or the rule policy won't be able to appropriately handle the orchestration message. Specifically, set the `Document Type` property of the business term to be the fully qualified type name—`SampleSolution.NewHireList` in this case. If the fully qualified .NET type name is not used to define the BizTalk schema, BizTalk Server will not have enough information to uniquely identify the data field.

■ **Note** In BizTalk Server 2004, the default value supplied for the `Document Type` property is the schema file name with no extension. If this value is not changed to use the fully qualified .NET type name, orchestrations will not be able to call the policy rules using the definition. You can find the fully qualified .NET type name by selecting a schema in the Visual Studio Solution Explorer and viewing the `Fully Qualified Name` property.

Figure 5-15 illustrates how to set the `Document Type` property for the `<Age>` element within the Business Rule Composer.

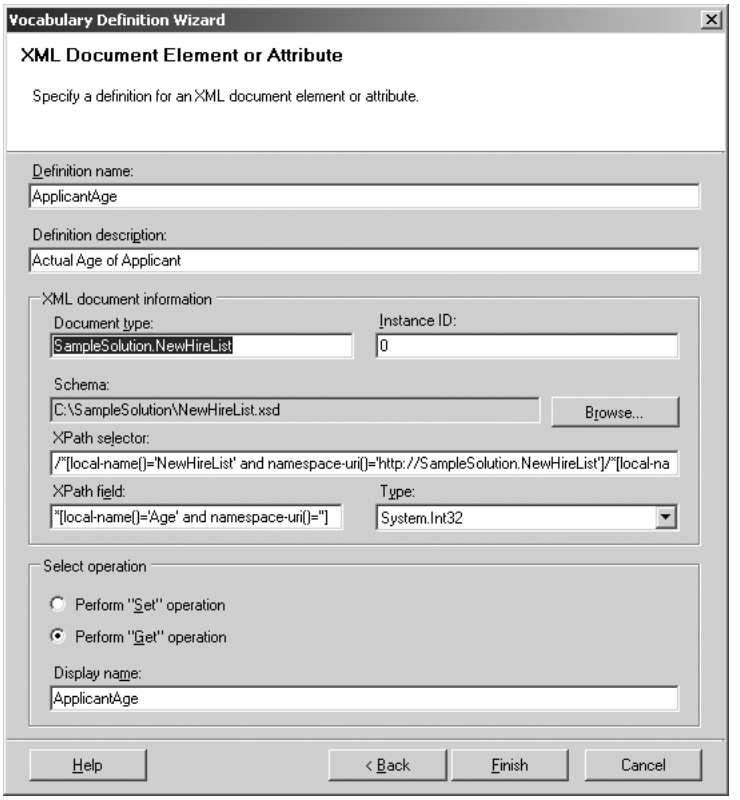

Figure 5-15. *Configuring the vocabulary definition*

A finer point regarding the Call Rules shape is that it treats input parameters as *reference* parameters, which means rules may modify the objects passed and return them to the orchestration. The solution in this chapter passes a new hire list message into the `SamplePolicy` policy. The policy modifies the message by applying the appropriate minimum age requirements logic and passes back the message. Given that messages are immutable (once they are created, they cannot be modified) in BizTalk, how does this work? The answer is that, behind the scenes, the policy creates a modified copy of the original message. Although this saves the developer's time

when creating their orchestration, it also has an important side effect: all promoted properties on the original message are removed from the message context. If properties on the message need to be maintained, make a copy of the message, allowing the message context to be reset after the business rules engine is called.

5-8. Deploying and Undeploying Policies

Problem

You want to know the components that make up a rules engine policy and perform the steps necessary to deploy and undeploy a policy to make the rule sets available to business processes.

Solution

Deploying a policy requires that several components be in place prior to the deployment. The deployment is quite straightforward, and you can deploy in several ways. This solution describes the prerequisites to the policy deployment and walks through the deployment steps using the Business Rule Composer. See the "How It Works" section for information about using the Rules Engine Deployment Wizard.

A policy consists of one or more rules. A rule comprises a set of facts. In this solution, we describe the process to deploy and undeploy all the components that make up a policy. Your first task is to define any facts that may need to be used in any rules that are part of the policy:

1. Define all the custom facts. Using the Business Rule Composer, open the Facts Explorer. Define any vocabulary, schemas, databases, or .NET assemblies that may be needed for any rule.

2. Once completed, save and publish any vocabularies that may have been defined. Right-click the vocabulary version, which will be used by a rule, and click Save. Right-click the version again, and select Publish. Once you have published a vocabulary version, you cannot modify it directly. Instead, it must be versioned—you can do this by copying the most recent version and then pasting a new version. The new version can be modified.

3. Your next task is to create a new policy and add all the rules that are part of the policy. In the Policy Explorer, create a new policy, and add one or more rules. As you create each rule, reference the appropriate version of the vocabularies, which have been published.

4. Save, publish, and test the policy by right-clicking the policy version and selecting the appropriate menu item.

5. The final task is to actually deploy the policy. Once the policy has been deployed, it cannot be deleted unless it is first undeployed. In the same manner as the vocabularies, it cannot be modified unless a new version is first created. Deploy the policy as follows: right-click the policy version that is to be deployed, and select Deploy, as shown in Figure 5-16. (You can undeploy policies by selecting Undeploy.)

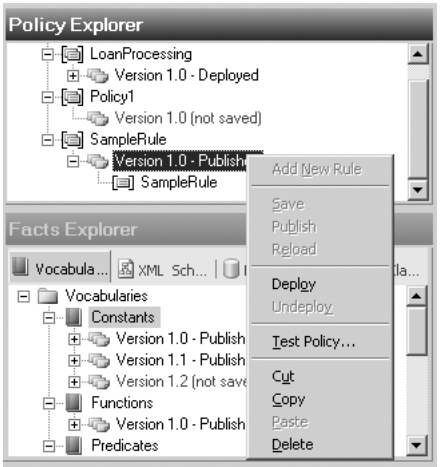

Figure 5-16. *Deploying via the Business Rule Composer*

How It Works

You have two basic approaches for deploying and undeploying policies. The first is to use the Business Rule Composer. Using the Business Rule Composer, you can define facts and policies that can be modified, published, and deployed. The second is to use the Rules Engine Deployment Wizard, as shown in Figure 5-17. The wizard allows policy files to be exported and imported onto machines where they may not have been created. Use the following basic steps to deploy a policy in a distributed environment:

1. Export the policy file. Open the BizTalk Rules Engine Deployment Wizard on the machine where the policy was created. Select Export Policy/Vocabulary File from Database. Walk through the options, selecting the policy you want to deploy on a target machine. This process will create an XML file (or multiple files if exporting multiple vocabularies and policies) that can be copied to another machine.

2. Copy the XML file(s) created in the first step to the target machine. Open the wizard on the target machine, and select the Import and Publish Policy/Vocabulary File to Database option. Walk through the options to publish the data to the target machine's rules engine (the vocabularies must first be deployed, and all referenced .NET assemblies must be placed in the GAC).

3. Run the wizard again on the target machine. This time, select the Deploy Policy option. All policies and vocabularies that have not been deployed will be available in the drop-down list. Select the object(s) you want to deploy, and walk through the rest of the wizard. The vocabularies and policies can now be referenced on the target machine by orchestrations or other components.

4. To undeploy, follow step 3, selecting Undeploy rather than Deploy from the first option list. Only those vocabularies and policies that have been deployed successfully will be available for undeployment.

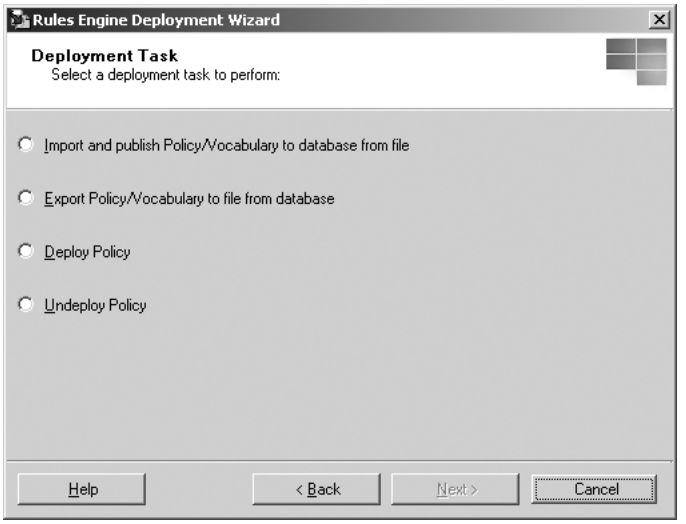

Figure 5-17. *Using the Rules Engine Deployment Wizard*