Microsoft®

Designing Data Tier Components and Passing Data Through Tiers

patterns & practices

proven practices for predictable results

Collaborators: Luca Bolognese, Mike Pizzo, Keith Short, Martin Petersen-Frey (PSS), Pablo De Grande, Bernard Chen (Sapient), Dimitris Georgakopoulos (Sapient), Kenny Jones, Chris Brooks, Lance Hendrix, Chris Schoon, and Franco Ceruti (VBNext).

Version 1.0

Contents

Introduction 1
Data Access Logic Components 2
Representing Business Entities 3
Technical Considerations 4
Mapping Relational Data to Business Entities 5
Recommendations for Mapping Relational Data to Business Entities 8
Implementing Data Access Logic Components 9
Application Scenarios for Data Access Logic Components 9
Implementing Data Access Logic Component Classes 11
Using Stored Procedures in Conjunction with Data Access Logic Components 14
Managing Locking and Concurrency 16
COM Interoperability 22
Implementing Business Entities 22
Representing Business Entities as XML 23
Representing Business Entities As a Generic DataSet 25
Representing Business Entities As a Typed DataSet 28
Defining Custom Business Entity Components 29
Defining Custom Business Entity Components with CRUD Behaviors 34
Recommendations for Representing Data and Passing Data Through Tiers 36
Transactions 37
Implementing Transactions 37
Recommendations for Using Manual Transactions in Data Access Logic Components 39
Recommendations for Using Automatic Transactions in Data Access Logic Components 39
Using Automatic Transactions in Business Entity Components 40
Validations 40
How to Validate XML by Using an XSD Schema 41
How to Validate Data in Property Accessors in Business Entity Components 42
Exception Management 43
Recommendations for Managing Exceptions in a Data Access Logic Component . . . 43
Recommendations for Managing Exceptions in Business Entity Components 44
Authorization and Security 45
Recommendations for Security in Data Access Logic Components 45
Recommendations for Security in Business Entity Components 48
Deployment 48
Deploying Data Access Logic Components 48
Deploying Business Entities 49

Appendix 50
Appendix Contents 50
How to Define a Data Access Logic Component Class 50
How to Use XML to Represent Collections and Hierarchies of Data 51
How to Apply a Style Sheet Programmatically in a .NET Application 52
How to Create a Typed DataSet 52
How to Define a Business Entity Component 54
How to Represent Collections and Hierarchies of Data in a Business Entity Component 55
How to Bind Business Entity Components to User-Interface Controls 57
How to Expose Events in a Business Entity Component 58
How to Serialize Business Entity Components to XML Format 60
How to Serialize Business Entity Components to SOAP Format 64
How to Serialize Business Entity Components to Binary Format 65

Introduction

When designing a distributed application, you need to decide how to access and represent the business data associated with your application. This document provides guidance to help you choose the most appropriate way of exposing, persisting, and passing that data through the tiers of an application.

Figure 1 depicts the common tiers of a distributed application. This document distinguishes between business data and the business processes that use the data; the business process tier is discussed only where needed for clarification. Likewise, the presentation tier is discussed only where there are direct implications for the way data is represented, such as the way ASP.NET Web pages expose business data. Figure 1 introduces two new terms: data access logic components and business entity components. These terms are described later in this document.

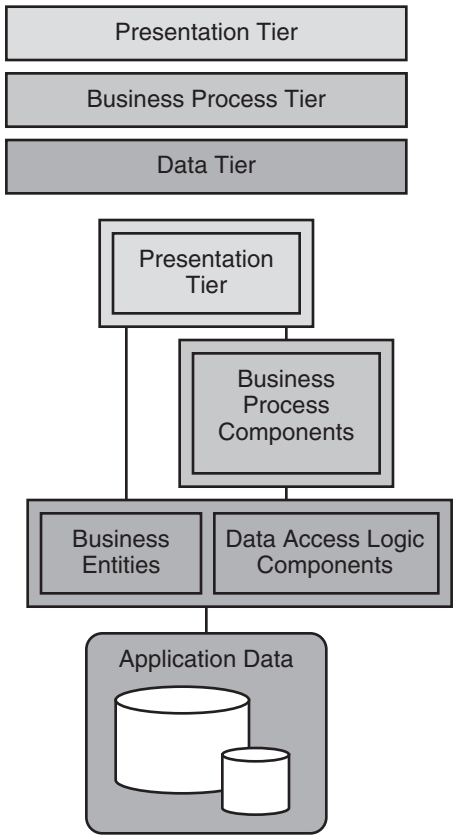

Figure 1

Accessing and representing data in a distributed application

Most applications store data in relational databases. While there are other options for storing data, this document focuses on how .NET applications interact with relational databases, and does not specifically discuss how to interact with data held in other data stores such as flat files or non-relational databases.

This document makes a clear distinction between persistence logic and the data itself. The reasons for separating persistence logic from the data include the following:

- Separate data persistence components can isolate the application from database dependencies, such as the name of the data source, connection information, and field names.
- Many of today's applications use loosely coupled, message-based technologies, such as XML Web services and Microsoft Message Queuing (also known as MSMQ). These applications typically communicate by passing business documents, rather than by passing objects.

Note: For an introduction to XML Web services, see the article titled .NET Web Services: Web Methods Make it Easy to Publish Your App's Interface over the Internet in the March 2002 issue of MSDN Magazine. For more information about Message Queuing, see Message Queuing Overview.

To attain the distinction between persistence logic and the data itself, this document proposes two different component types.

- Data access logic components. Data access logic components retrieve data from the database and save entity data back to the database. Data access logic components also contain any business logic needed to achieve data-related operations.
- Business entity components. Data is used to represent real world business entities, such as products or orders. There are numerous ways to represent these business entities in your application—for example, XML or DataSets or custom object-oriented classes—depending on the physical and logical design constraints of the application. Design options are investigated in detail later in this document.

Data Access Logic Components

A Data Access Logic Component provides methods to perform the following tasks upon a database, on behalf of the caller:

- Create records in the database.
- Read records in the database, and return business entity data to the caller.
- Update records in the database, by using revised business entity data supplied by the caller.
- Delete records in the database.

The methods that perform the preceding tasks are often called "CRUD" methods, where CRUD is an acronym based on the first letter of each task.

The Data Access Logic Component also has methods to implement business logic against the database. For example, a Data Access Logic Component might have a method to find the highest-selling product in a catalog for this month.

Typically, a Data Access Logic Component accesses a single database and encapsulates the data-related operations for a single table or a group of related tables in the database. For example, you might define one Data Access Logic Component to deal with the Customer and Address tables in a database, and another Data Access Logic Component to deal with the Orders and OrderDetails tables. The design decisions for mapping data access logic components to database tables are discussed later in this document.

Representing Business Entities

Each Data Access Logic Component deals with a specific type of business entity. For example, the Customer Data Access Logic Component deals with Customer business entities. There are many different ways to represent business entities, depending on factors such as the following:

- Do you need to bind business entity data to controls in a Windows form or on an ASP.NET page?
- Do you need to perform sorting or searching operations on the business entity data?
- Does your application deal with business entities one at a time, or does it typically deal with sets of business entities?
- Will you deploy your application locally or remotely?
- Will the business entity be used by XML Web services?
- How important are nonfunctional requirements, such as performance, scalability, maintainability, and programming convenience?

This document outlines the advantages and disadvantages of the following implementation options:

- XML. You use an XML string or an XML Document Object Model (DOM) object to represent business entity data. XML is an open and flexible data representation format that can be used to integrate diverse types of applications.
- DataSet. A DataSet is an in-memory cache of tables, obtained from a relational database or an XML document. A Data Access Logic Component can use a DataSet to represent business entity data retrieved from the database, and you can use the DataSet in your application. For an introduction to DataSets, see "Introducing ADO.NET" in the .NET Data Access Architecture Guide.

- Typed DataSet. A typed DataSet is a class that inherits from the ADO.NET DataSet class and provides strongly typed methods, events, and properties to access the tables and columns in a DataSet.
- Business Entity Component. This is a custom class to represent each type of business entity. You define fields to hold the business entity data, and you define properties to expose this data to the client application. You define methods to encapsulate simple business logic, making use of the fields defined in the class. This option does not implement CRUD methods as pass-through methods to the underlying Data Access Logic Component; the client application communicates directly with the Data Access Logic Component to perform CRUD operations.
- Business Entity Component with CRUD behaviors. You define a custom entity class as described previously, and you implement the CRUD methods that call the underlying Data Access Logic Component associated with this business entity.

Note: If you prefer to work with your data in a more object-oriented fashion, you can use the alternate approach of defining an object persistence layer based on the reflection capabilities of the common language runtime. You can create a framework that uses reflection to read the properties of the objects and use a mapping file to describe the mapping between objects and tables. However, to implement this effectively would constitute a major investment in infrastructure code. This outlay might be viable for ISVs and solution providers, but not for the majority of organizations, and it is beyond the scope of this document.

Technical Considerations

Figure 2 shows some of the technical considerations that influence the implementation strategy for data access logic components and business entities. This document addresses each of these technical considerations and provides recommendations.

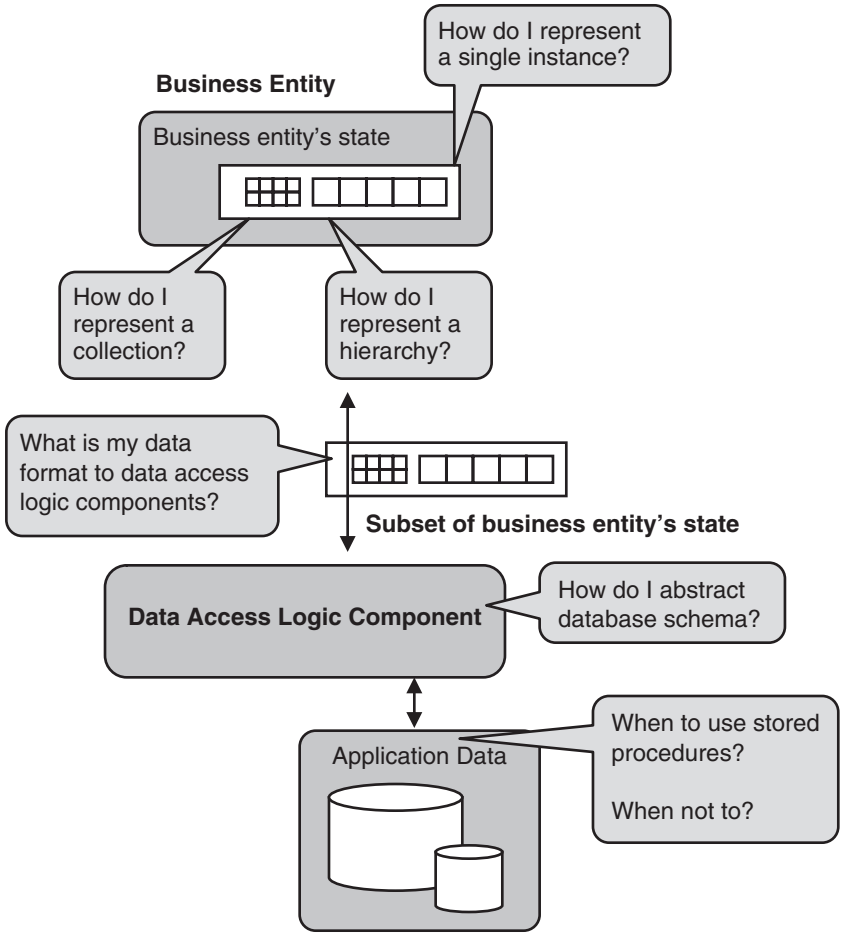

Figure 2

Technical considerations that influence the design of data access logic components and business entities

Mapping Relational Data to Business Entities

Databases typically contain many tables, with relationships implemented by primary keys and foreign keys in these tables. When you define business entities to represent this data in your .NET application, you must decide how to map these tables to business entities.

Consider the hypothetical retailer's database shown in Figure 3.

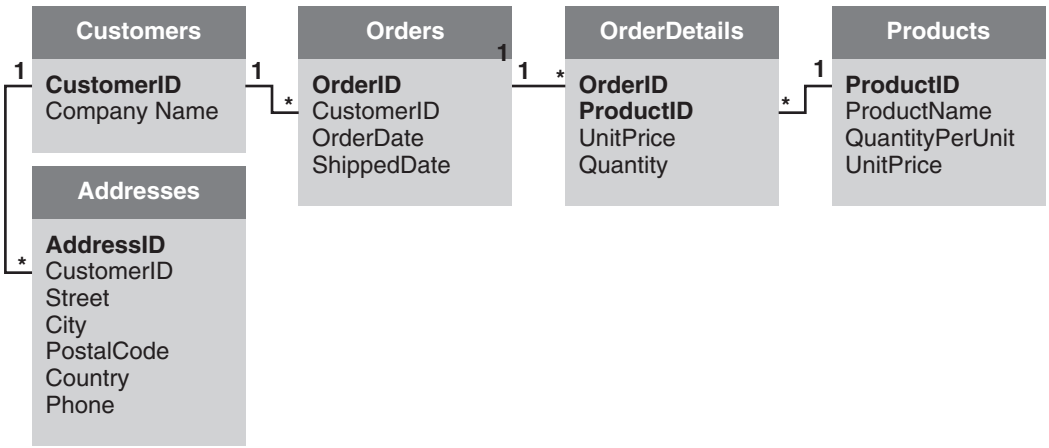

Figure 3
Hypothetical table relationships in a relational database

The following table summarizes the types of relationships in the example database.

Type of relationship	Example	Description
One-to-many	Customer: Address Customer: Order	A customer can have many addresses, such as a delivery address, a billing address, and a contact address. A customer can place several orders.
Many-to-many	Order: Product	An order can comprise many products; each product is represented by a separate row in the OrderDetails table. Likewise, a product can be featured in many orders.

When you define business entities to model the information in the database, consider how you will use the information in your application. Identify the core business entities that encapsulate your application's functionality, rather than defining a separate business entity for each table.

Typical operations in the hypothetical retailer's application are as follows:

- Get (or update) information about a customer, including his or her addresses.
- Get a list of orders for a customer.
- Get a list of order items for a particular order.
- Place a new order.
- Get (or update) information about a product or a collection of products.

To fulfill these application requirements, there are three logical business entities that the application will handle: a Customer, an Order, and a Product. For each business entity, a separate Data Access Logic Component will be defined, as follows:

- Customer Data Access Logic Component. This class will provide services to retrieve and modify data in the Customer and Address tables.
- Order Data Access Logic Component. This class will provide services to retrieve and modify data in the Order and OrderDetails tables.
- Product Data Access Logic Component. This class will provide services to retrieve and modify data in the Product table.

Figure 4 illustrates the relationships between the data access logic components and the tables that they represent in the database.

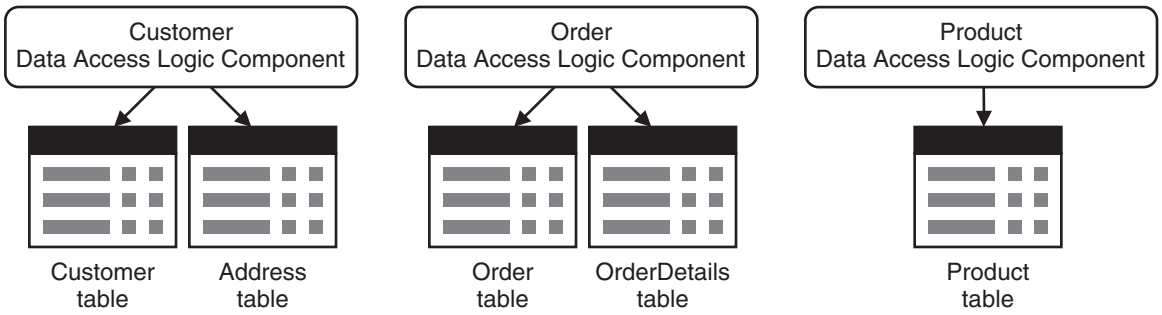

Figure 4

Defining data access logic components to expose relational data to .NET applications

For a description of how to implement data access logic components, see Implementing Data Access Logic Components later in this document.

Recommendations for Mapping Relational Data to Business Entities

To map relational data to business entities, consider the following recommendations:

- Take the time to analyze and model the logical business entities of your application, rather than defining a separate business entity for every table. One of the ways to model how your application works is to use Unified Modeling Language (UML). UML is a formal design notation for modeling objects in an object-oriented application, and for capturing information about how objects represent automated processes, human interactions, and associations. For more information, see Modeling Your Application and Data.
- Do not define separate business entities to represent many-to-many tables in the database; these relationships can be exposed through methods implemented in your Data Access Logic Component. For example, the OrderDetails table in the preceding example is not mapped to a separate business entity; instead, the Orders data access logic component encapsulates the OrderDetails table to achieve the many-to-many relationship between the Order and Product tables.
- If you have methods that return a particular type of business entity, place these methods in the Data Access Logic Component for that type. For example, if you are retrieving all orders for a customer, implement that function in the Order Data Access Logic Component because your return value is of the type Order. Conversely, if you are retrieving all customers that have ordered a specific product, implement that function in the Customer Data Access Logic Component.
- Data access logic components typically access data from a single data source. If aggregation from multiple data sources is required, it is recommended to define a separate Data Access Logic Component to access each data source that can be called from a higher-level business process component that can perform the aggregation. There are two reasons for this recommendation:
  - Transaction management is centralized to the business process component and does not need to be controlled explicitly by the Data Access Logic Component. If you access multiple data sources from one Data Access Logic Component, you will need the Data Access Logic Component to be the root of transactions, which will introduce additional overhead on functions where you are only reading data.
  - Aggregation is usually not a requirement in all areas of the application, and by separating the access to the data, you can let the type stand alone as well as be part of an aggregation when needed.

Implementing Data Access Logic Components

A Data Access Logic Component is a stateless class, meaning that all messages exchanged can be interpreted independently. No state is held between calls. The Data Access Logic Component provides methods for accessing one or more related tables in a single database, or in some instances, multiple databases as in the case of horizontal database partitioning. Typically, the methods in a Data Access Logic Component invoke stored procedures to perform their operations.

One of the key goals of data access logic components is to hide the invocation and format idiosyncrasies of the database from the calling application. Data access logic components provide an encapsulated data-access service to these applications. Specifically, data access logic components handle the following implementation details:

- Manage and encapsulate locking schemes
- Handle security and authorization issues appropriately
- Handle transaction issues appropriately
- Perform data paging
- Perform data-dependent routing if required
- Implement a caching strategy if appropriate, for queries of nontransactional data
- Perform data streaming and data serialization

Some of these issues are explored in more detail later in this section.

Application Scenarios for Data Access Logic Components

Figure 5 on the next page shows how a Data Access Logic Component can be called from a variety of application types, including Windows Forms applications, ASP.NET applications, XML Web services, and business processes. These calls might be local or remote, depending on how you deploy your applications.

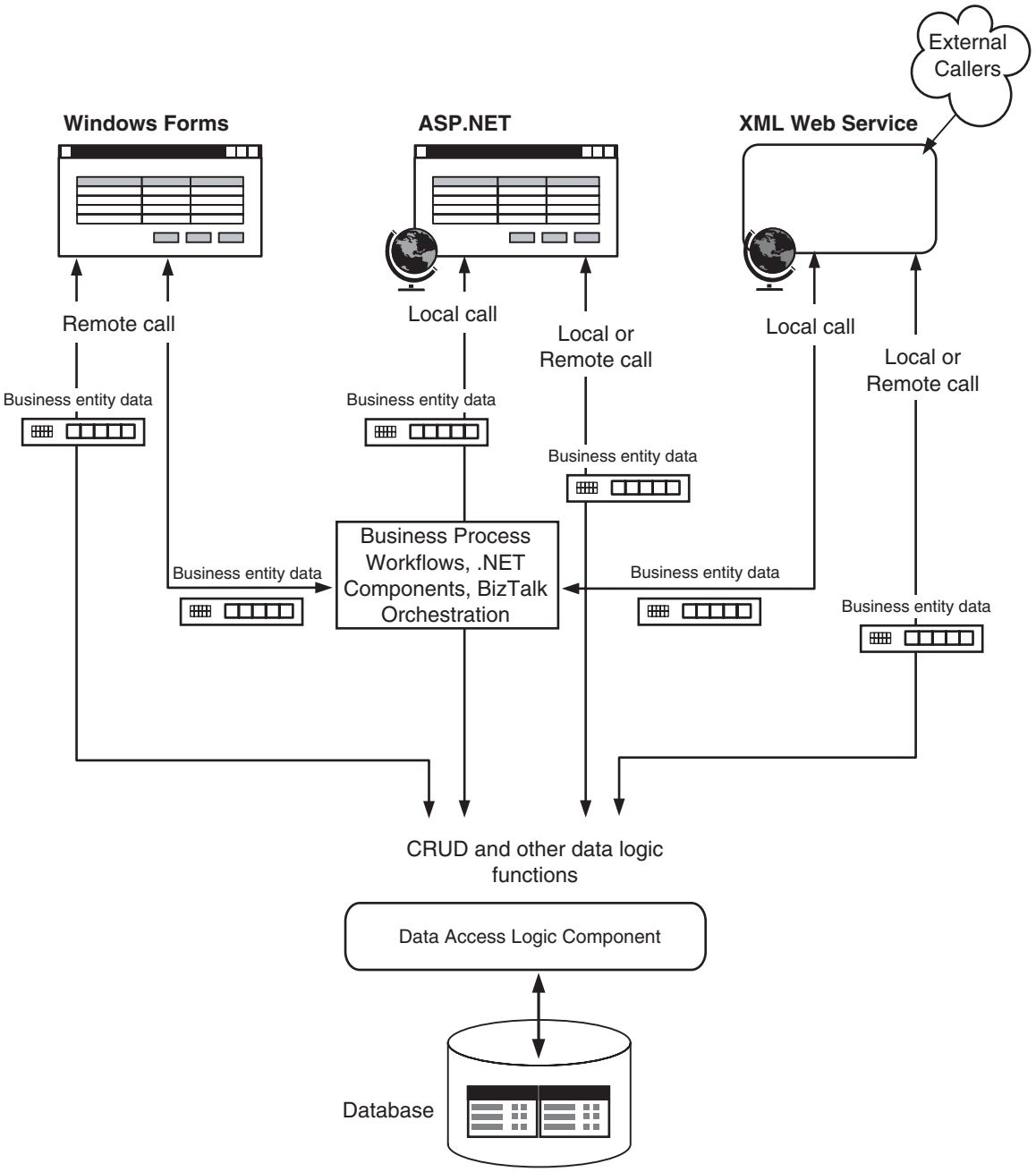

Figure 5

Application scenarios for data access logic components

Implementing Data Access Logic Component Classes

Data access logic components use ADO.NET to execute SQL statements or call stored procedures. For an example of a Data Access Logic Component class, see How to Define a Data Access Logic Component Class in the appendix.

If your application contains multiple data access logic components, you can simplify the implementation of Data Access Logic Component classes by using a data access helper component. This component can help manage database connections, execute SQL commands, and cache parameters. The data access logic components still encapsulate the logic required to access the specific business data, whereas the data access helper component centralizes data access API development and data connection configuration, thereby helping to reduce code duplication. Microsoft provides the Data Access Application Block for .NET, which can be used as a generic data access helper component in your applications when you use Microsoft SQL Server™ databases. Figure 6 shows how to use the data access helper component to help implement data access logic components.

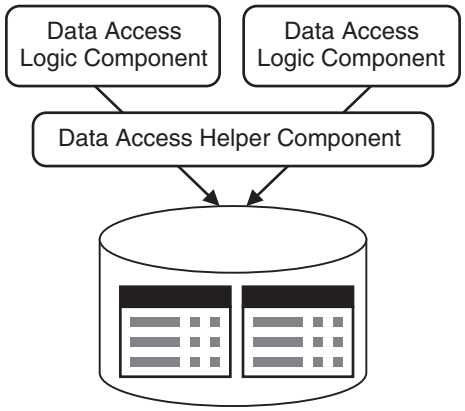

Figure 6
Implementing data access logic components by using the data access helper component

If there are utility functions that are common to all of your data access logic components, you can define a base class for data access logic components to inherit from and extend.

Design your Data Access Logic Component classes to provide a consistent interface for different types of clients. If you design the Data Access Logic Component to be compatible with the requirements of your current and potential business process tier implementation, you can reduce the number of additional interfaces, façades, or mapping layers that you must implement.

To support a diverse range of business processes and applications, consider the following techniques to pass data to and from Data Access Logic Component methods:

- Passing business entity data into methods in the Data Access Logic Component. You can pass the data in several different formats: as a series of scalar values, as an XML string, as a DataSet, or as a custom Business Entity Component.
- Returning business entity data from methods in the Data Access Logic Component. You can return the data in several different formats: as output-parameter calar values, as an XML string, as a DataSet, as a custom Business Entity Component, or as a data reader.

The following sections present the options for passing business entity data to and from your data access logic components, in addition to the advantages and disadvantages of each approach. This information will help you to make an informed choice based on your specific application scenario.

Passing Scalar Values As Inputs and Outputs

The advantages of this option are as follows:

- Abstraction. Callers must know about only the data that defines the business entity, but not a specific type or the specific structure of the business entity.
- Serialization. Scalar values natively support serialization.
- Efficient use of memory. Scalar values only convey the data that is actually needed.
- Performance. When dealing with instance data, scalar values offer better performance than the other options described in this document.

The disadvantages of this option are as follows:

- Tight coupling and maintenance. Schema changes could require method signatures to be modified, which will affect the calling code.
- Collections of entities. To save or update multiple entities to a Data Access Logic Component, you must make separate method calls. This can be a significant performance hit in distributed environments.
- Support of optimistic concurrency. To support optimistic concurrency, time stamp columns must be defined in the database and included as part of the data.

Passing XML Strings As Inputs and Outputs

The advantages of this option are as follows:

- Loose coupling. Callers must know about only the data that defines the business entity and the schema that provides metadata for the business entity.

- Integration. Accepting XML will support callers implemented in various ways—for example, .NET applications, BizTalk Orchestration rules, and third-party business rules engines.
- Collections of business entities. An XML string can contain data for multiple business entities.
- Serialization. Strings natively support serialization.

The disadvantages of this option are as follows:

- Reparsing effort for XML strings. The XML string must be reparsed at the receiving end. Very large XML strings incur a performance overhead.
- Inefficient use of memory. XML strings can be verbose, which can cause inefficient use of memory if you need to pass large amounts of data.
- Supporting optimistic concurrency. To support optimistic concurrency, time stamp columns must be defined in the database and included as part of the XML data.

Passing DataSets As Inputs and Outputs

The advantages of this option are as follows:

- Native functionality. DataSets provide built-in functionality to handle optimistic concurrency (along with data adapters) and support for complex data structures. Furthermore, typed DataSets provide support for data validation.
- Collections of business entities. DataSets are designed to handle sets and complex relationships, so you do not need to write custom code to implement this functionality.
- Maintenance. Schema changes do not affect the method signatures. However, if you are using typed DataSets and the assembly has a strong name, the Data Access Logic Component class must be recompiled against the new version, must use a publisher policy inside the global assembly cache, or must define a <bindingRedirect> element in its configuration file. For information about how the runtime locates assemblies, see How the Runtime Locates Assemblies.
- Serialization. A DataSet supports XML serialization natively and can be serialized across tiers.

The disadvantages of this option are as follows:

- Performance. Instantiating and marshalling DataSets incur a runtime overhead.
- Representation of a single business entity. DataSets are designed to handle sets of data. If your application works mainly with instance data, scalar values or custom entities are a better approach as you will not incur the performance overhead.

Passing Custom Business Entity Components As Inputs and Outputs

The advantages of this option are as follows:

- Maintenance. Schema changes may not affect the Data Access Logic Component method signatures. However, the same issues arise as with typed DataSets if the Business Entity Component is held in a strong-named assembly.
- Collections of business entities. An array or a collection of custom business entity components can be passed to and from the methods.

The disadvantages of this option are as follows:

- Supporting optimistic concurrency. To support optimistic concurrency easily, time stamp columns must be defined in the database and included as part of the instance data.
- Limited integration. When using custom business entity components as inputs to the Data Access Logic Component, the caller must know the type of business entity; this can limit integration for callers that are not using .NET. However, this issue does not necessarily limit integration if the caller uses custom business entity components as output from the Data Access Logic Component. For example, a Web method can return the custom Business Entity Component that was returned from a Data Access Logic Component, and the Business Entity Component will be serialized to XML automatically using XML serialization.

Returning Data Readers As Outputs

The advantage of this option is as follows:

- Performance. There is a performance benefit when you need to render data quickly and you can deploy your Data Access Logic Component with the presentation tier code.

The disadvantage of this option is as follows:

- Remoting. It is inadvisable to use data readers in remoting scenarios, because of the potential for client applications to hold the database connection open for lengthy periods.

Using Stored Procedures in Conjunction with Data Access Logic Components

You can use stored procedures to perform many of the data access tasks supported by data access logic components.

Advantages

- Stored procedures generally result in improved performance, because the database can optimize the data access plan used by the procedure and cache the plan for subsequent reuse.

- Stored procedures can be individually secured within the database. An administrator can grant clients permission to execute a stored procedure, without granting any permissions on the underlying tables.
- Stored procedures may result in easier maintenance, because it is generally easier to modify a stored procedure than to change a hard-coded SQL statement within a deployed component. However, the benefit decreases as the business logic implemented in the stored procedures increases.
- Stored procedures add an extra level of abstraction from the underlying database schema. The client of the stored procedure is isolated from the implementation details of the stored procedure and from the underlying schema.
- Stored procedures can reduce network traffic. SQL statements can be executed in batches, rather than the application having to send multiple SQL requests.

Despite the advantages listed above, there are some situations where the use of stored procedures is not recommended or may be infeasible.

Disadvantages

- Applications that involve extensive business logic and processing could place an excessive load on the server if the logic was implemented entirely in stored procedures. Examples of this type of processing include data transfers, data traversals, data transformations, and intensive computational operations. You should move this type of processing to business process or data access logic components, which are a more scalable resource than your database server.
- Do not put all of your business logic into stored procedures. Maintenance and the agility of your application becomes an issue when you must modify business logic in T-SQL. For example, ISV applications that support multiple RDBMS should not need to maintain separate stored procedures for each system.
- Writing and maintaining stored procedures is most often a specialized skill set that not all developers possess. This situation may introduce bottlenecks in the project development schedule.

Recommendations for Using Stored Procedures with Data Access Logic Components

Consider the following recommendations for using stored procedures in conjunction with data access logic components:

- Exposing stored procedures. Data access logic components should be the only components that are exposed to database schema information, such as stored procedure names, parameters, tables, and fields. Your business entity implementation should have no knowledge of or dependency on database schemas.
- Associating stored procedures with data access logic components. Each stored procedure should be called by only one Data Access Logic Component, and

should be associated with the Data Access Logic Component that owns the action. For example, imagine that a customer places an order with a retailer. You can write a stored procedure named OrderInsert, which creates the order in the database. In your application, you must decide whether to call the stored procedure from the Customer Data Access Logic Component or from the Order Data Access Logic Component. The Order Data Access Logic Component is a better choice because it handles all order-related processing (the Customer Data Access Logic Component handles customer information, such as the customer's name and address).

- Naming stored procedures. When you define stored procedures for a Data Access Logic Component to use, choose stored procedure names that emphasize the Data Access Logic Component to which they pertain. This naming convention helps to easily identify which components call which stored procedures, and provides a way to logically group the stored procedures inside SQL Enterprise Manager. For example, you can proactively write stored procedures named CustomerInsert, CustomerUpdate, CustomerGetByCustomerID, and CustomerDelete for use by the Customer Data Access Logic Component, and then provide more specific stored procedures — such as CustomerGetAllInRegion —to support the business functions of your application.

  Note: Do not preface your stored procedure names with sp_, because doing so reduces performance. When you call a stored procedure that starts with sp_, SQL Server always checks the master database first, even if the stored procedure is qualified with the database name.

- Addressing security issues. If you accept user input to perform queries dynamically, do not create a string by concatenating values without using parameters. Also avoid using string concatenation in stored procedures if you are using sp_execute to execute the resulting string, or if you do not take advantage of sp_executesql parameter support.

Managing Locking and Concurrency

Some applications take the "Last in Wins" approach when it comes to updating data in a database. With the "Last in Wins" approach, the database is updated, and no effort is made to compare updates against the original record, potentially overwriting any changes made by other users since the records were last refreshed. However, at times it is important for the application to determine if the data has been changed since it was initially read, before performing the update.

Data access logic components implement the code to manage locking and concurrency. There are two ways to manage locking and concurrency:

- Pessimistic concurrency. A user who reads a row with the intention of updating it establishes a lock on the row in the data source. No one else can change the row until the user releases the lock.
- Optimistic concurrency. A user does not lock a row when reading it. Other users are free to access the row in the meantime. When a user wants to update a row, the application must determine whether another user has changed the row since it was read. Attempting to update a record that has already been changed causes a concurrency violation.

Using Pessimistic Concurrency

Pessimistic concurrency is primarily used in environments where there is heavy contention for data, and where the cost of protecting data through locks is less than the cost of rolling back transactions if concurrency conflicts occur. Pessimistic concurrency is best implemented when lock times will be short, as in programmatic processing of records.

Pessimistic concurrency requires a persistent connection to the database and is not a scalable option when users are interacting with data, because records might be locked for relatively large periods of time.

Using Optimistic Concurrency

Optimistic concurrency is appropriate in environments where there is low contention for data, or where read-only access to data is required. Optimistic concurrency improves database performance by reducing the amount of locking required, thereby reducing the load on the database server.

Optimistic concurrency is used extensively in .NET to address the needs of mobile and disconnected applications, where locking data rows for prolonged periods of time would be infeasible. Also, maintaining record locks requires a persistent connection to the database server, which is not possible in disconnected applications.

Testing for Optimistic Concurrency Violations

There are several ways to test for optimistic concurrency violations:

- Use distributed time stamps. Distributed time stamps are appropriate when reconciliation is not required. Add a time stamp or version column to each table in the database. The time stamp column is returned with any query of the contents of the table. When an update is attempted, the time stamp value in the database is compared to the original time stamp value contained in the modified row. If the values match, the update is performed and the time stamp column is updated with the current time to reflect the update. If the values do not match, an optimistic concurrency violation has occurred.

- Maintain a copy of original data values. When you query data from the database, keep a copy of any original data values. When you update the database, verify that the current values in the database match the original values.
- DataSets hold original values that the data adapter can use to perform optimistic concurrency checks when you update the database.
- Use centralized time stamps. Define a centralized time stamp table in your database, to log all updates to any row in any table. For example, the time stamp table can indicate the following: "Row with ID 1234 on table XYZ was updated by John on March 26, 2002 2:56 P.M."

  Centralized time stamps are appropriate in checkout scenarios and in some disconnected client scenarios where clear owners and management of locks and overrides may be needed. In addition, centralized time stamps provide the benefit of auditing when needed.

Manually Implementing Optimistic Concurrency

Consider the following SQL query:

```
SELECT Column1, Column2, Column3 FROM Table1
```

To test for an optimistic concurrency violation when updating a row in Table1, issue the following UPDATE statement:

```
UPDATE Table1 Set Column1 = @NewValueColumn1,
              Set Column2 = @NewValueColumn2,
              Set Column3 = @NewValueColumn3
WHERE Column1 = @OldValueColumn1 AND
      Column2 = @OldValueColumn2 AND
      Column3 = @OldValueColumn3
```

If the original values match the values in the database, the update is performed. If a value has been modified, the update will not modify the row because the WHERE clause will not find a match. You can use a variation of this technique and apply a WHERE clause only to specific columns, resulting in data being overwritten unless particular fields have been updated since they were last queried.

Note: Always return a value that uniquely identifies a row in your query, such as a primary key, to use in your WHERE clause of the UPDATE statement. This ensures that the UPDATE statement updates the correct row or rows.

If a column at your data source allows nulls, you may need to extend your WHERE clause to check for a matching null reference in your local table and at the data source. For example, the following UPDATE statement verifies that a null reference in the local row still matches a null reference at the data source, or that the value in the local row still matches the value at the data source:

```
UPDATE Table1 Set Column1 = @NewColumn1Value
WHERE (@OldColumn1Value IS NULL AND Column1 IS NULL) OR Column1 = @OldColumn1Value
```

Using Data Adapters and DataSets to Implement Optimistic Concurrency

The DataAdapter.RowUpdated event can be used in conjunction with the techniques described earlier to provide notification to your application of optimistic concurrency violations. The RowUpdated event occurs after each attempt to update a modified row from a DataSet. You can use the RowUpdated event to add special handling code, including processing when an exception occurs, adding custom error information, and adding retry logic.

The RowUpdated event handler receives a RowUpdatedEventArgs object, which has a RecordsAffected property that indicates how many rows were affected by an update command for a modified row in a table. If you set the update command to test for optimistic concurrency, the RecordsAffected property will be 0 when an optimistic concurrency violation occurs. Set the RowUpdatedEventArgs.Status property to indicate how to proceed; for example, set the property to UpdateStatus.SkipCurrentRow to skip updating the current row, but to continue updating the other rows in the update command. For more information about the RowUpdated event, see Working with DataAdapter Events.

An alternative way to test for concurrency errors with a data adapter is to set the DataAdapter.ContinueUpdateOnError property to true before you call the Update method. When the update is completed, call the GetErrors method on the DataTable object to determine which rows have errors. Then, use the RowError property on these rows to find specific error details. For more information about how to process row errors, see Adding and Reading Row Error Information.

The following code sample shows how the Customer Data Access Logic Component can check for concurrency violations. This example assumes that the client has retrieved a DataSet, made changes to the data, and then passed the DataSet to the UpdateCustomer method on the Data Access Logic Component. The UpdateCustomer method will invoke the following stored procedure to update the appropriate customer record; the stored procedure updates the record only if the customer ID and company name have not already been modified:

```
CREATE PROCEDURE CustomerUpdate
{
  @CompanyName varchar(30),
  @oldCustomerID varchar(10),
  @oldCompanyName varchar(30)
}
AS
  UPDATE Customers Set CompanyName = @CompanyName
  WHERE CustomerID = @oldCustomerID AND CompanyName = @oldCompanyName
GO
```

Inside the UpdateCustomer method, the following code sample sets the UpdateCommand property of a data adapter to test for optimistic concurrency, and then uses the RowUpdated event to test for optimistic concurrency violations. If an optimistic concurrency violation is encountered, the application indicates the violation by setting the RowError of the row for which the update was issued. Note that the parameter values passed to the WHERE clause of the UPDATE command are mapped to the Original values of the respective columns in the DataSet.

```
// UpdateCustomer method in the CustomerDALC class
public void UpdateCustomer(DataSet dsCustomer)
{
  // Connect to the Northwind database
  SqlConnection cnNorthwind = new SqlConnection(
    "Data source=localhost;Integrated security=SSPI;Initial Catalog=northwind");

  // Create a Data Adapter to access the Customers table in Northwind
  SqlDataAdapter da = new SqlDataAdapter();

  // Set the Data Adapter's UPDATE command, to call the "UpdateCustomer" stored
  // procedure
  da.UpdateCommand = new SqlCommand("CustomerUpdate", cnNorthwind);
  da.UpdateCommand.CommandType = CommandType.StoredProcedure;

  // Add two parameters to the Data Adapter's UPDATE command, to specify
  // information for the WHERE clause (to facilitate checking for optimistic
  // concurrency violation)
  da.UpdateCommand.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 30,
"CompanyName");

  // Specify the original value of CustomerID as the first WHERE clause parameter
  SqlParameter myParm = da.UpdateCommand.Parameters.Add(
                            "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
  myParm.SourceVersion = DataRowVersion.Original;

  // Specify the original value of CustomerName as the second WHERE clause
  // parameter
  myParm = da.UpdateCommand.Parameters.Add(
                            "@oldCompanyName", SqlDbType.NVarChar, 30,
"CompanyName");
  myParm.SourceVersion = DataRowVersion.Original;

  // Add a handler for Row Update events
  da.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

  // Update the database
  da.Update(ds, "Customers");
```

```
  foreach (DataRow myRow in ds.Tables["Customers"].Rows)
  {
    if (myRow.HasErrors)
      Console.WriteLine(myRow[0] + " " + myRow.RowError);
  }
}

// Method to handle Row Update events. If you register the event but do not handle
// it, a SQL exception is thrown.
protected static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs args)
{
  if (args.RecordsAffected == 0)
  {
    args.Row.RowError = "Optimistic Concurrency Violation Encountered";
    args.Status = UpdateStatus.SkipCurrentRow;
  }
}
```

When executing multiple SQL statements within one SQL Server stored procedure, you may, for performance reasons, choose to use the SET NOCOUNT ON option. This option prevents SQL Server from returning a message to the client after completion of each statement, thus reducing network traffic. However, you will not be able to check the RecordsAffected property as demonstrated in the previous code sample. The RecordsAffected property will always be –1. An alternative is to return the @@ROWCOUNT function (or specify it as an output parameter) in your stored procedure; @@ROWCOUNT contains the record count for the last statement completed in your stored procedure and is updated even when SET NOCOUNT ON is used. Therefore, if the last SQL statement executed in your stored procedure is the actual UPDATE statement and you specify the @@ROWCOUNT as a return value, you can modify application code as follows:

```
// Add another parameter to Data Adapter's UPDATE command, to accept the return
// value. You can name it anything you want.
myParm = da.UpdateCommand.Parameters.Add("@RowCount", SqlDbType.Int);
myParm.Direction = ParameterDirection.ReturnValue;

// Modify the OnRowUpdated method, to check the value of this parameter
// instead of the RecordsAffected property.
protected static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs args)
{
  if (args.Command.Parameters["@RowCount"].Value == 0)
  {
    args.Row.RowError = "Optimistic Concurrency Violation Encountered";
    args.Status = UpdateStatus.SkipCurrentRow;
  }
}
```

COM Interoperability

If you want your Data Access Logic Component class to be callable from COM clients, the recommended approach is to define your data access logic components by using the preceding guidelines and provide a wrapper component. However, if you want COM clients to be able to access your data access logic components, consider the following recommendations:

- Define the class and its members as public.
- Avoid using static members.
- Define event-source interfaces in managed code.
- Provide a constructor that does not use parameters.
- Do not use overloaded methods. Use methods with different names instead.
- Use interfaces to expose common operations.
- Use attributes to provide extra COM information for your class and members.
- Include HRESULT values in any exceptions thrown from .NET code.
- Use automation-compatible data types in method signatures.

For more information about COM interoperability, see the Microsoft .NET/COM Migration and Interoperability guide.

Implementing Business Entities

Business entities exhibit the following characteristics:

- Business entities provide stateful programmatic access to business data and (in some designs) related functionality.
- Business entities can be built from data that has complex schemas. The data typically originates from multiple related tables in the database.
- Business entity data can be passed as part of the I/O parameters of business processes.
- Business entities can be serializable, to persist the current state of the entities. For example, applications may need to store entity data on a local disk, in a desktop database if the application is working offline, or in a Message Queuing message.
- Business entities do not access the database directly. All database access is provided by the associated Data Access Logic Component.
- Business entities do not initiate any kind of transaction. Transactions are initiated by the application or business process that is using the business entities.

As mentioned earlier in this document, there are various ways to represent business entities in your application, ranging from a data-centric model to a more object-oriented representation:

- XML
- Generic DataSet
- Typed DataSet
- Custom business entity components
- Custom business entity components with CRUD behaviors

The sections that follow describe how to represent business entity data in each of these formats. To help you decide the most appropriate representation for business entities in your particular circumstances, the sections describe how to perform the following tasks for each business entity format:

- Organize collections of business entities
- Data bind business entities to user interface controls
- Serialize business entity data
- Pass business entity data between tiers

The sections also consider the suitability of each business entity representation in terms of nonfunctional requirements, including performance, efficiency, scalability, and extensibility.

Representing Business Entities as XML

The following example shows how to represent a simple business entity as XML. The business entity consists of a single product.

```
<?xml version="1.0"?>
<Product xmlns="urn:aUniqueNamespace">
  <ProductID>1</ProductID>
  <ProductName>Chai</ProductName>
  <QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
  <UnitPrice>18.00</UnitPrice>
  <UnitsInStock>39</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>10</ReorderLevel>
</Product>
```

For more information, see How to Use XML to Represent Collections and Hierarchies of Data in the appendix.

When you use XML to represent business entity data, consider the following guidelines:

- Decide whether the XML document should contain a single business entity or a collection of business entities. The preceding example represents a single Product business entity.
- Use a namespace to uniquely identify the XML document, to avoid name clashes with content in other XML documents. The preceding example uses a default namespace called urn:aUniqueNamespace.
- Choose appropriate names for elements and attributes. The preceding example uses the column names from the Product table, but this is not a requirement. Choose names that make sense for your application.
- Use one of the following approaches to retrieve your business entities in XML format:
  - If you are using SQL Server 2000, you can use the FOR XML clause in your queries or stored procedures. In performance tests, using FOR XML is only slightly faster than returning a DataSet.
  - Retrieve a DataSet and transform it or write it out as an XML stream. With this approach, you incur the overhead of creating the DataSet and additional overhead if performing a transformation.
  - Build an XML document from output parameters or by using a data reader. Data readers are the fastest way to retrieve multiple rows from the database, but the process associated with building XML may decrease the performance benefit.

For more information and performance considerations, see Performance Comparison: Data Access Techniques.

The advantages of representing business entities as XML are as follows:

- Standards support. XML is a World Wide Web Consortium (W3C) standard data representation format. For more information about this standard, see *http://www.w3.org/xml*.
- Flexibility. XML can represent hierarchies and collections of information. For more information, see How to Use XML to Represent Collections and Hierarchies of Data in the appendix.
- Interoperability. XML is an ideal choice for exchanging information with external parties and trading partners, regardless of platform. If the XML data will be consumed by ASP.NET or Windows Forms applications, you can load the XML data into a DataSet to take advantage of the data binding support provided by DataSets.

The disadvantages of representing business entities as XML are as follows:

- Preserving type fidelity. Type fidelity is not preserved in XML. However, you can use XSD schemas for simple data typing.
- Validating XML. To validate XML, you can parse the code manually or use an XSD schema. Both approaches are relatively slow. For an example of how to validate XML by using an XSD schema, see How to Validate XML by Using an XSD Schema.
- Displaying XML. You cannot automatically display XML data in the user interface. You can write an XSLT style sheet to transform the data into a DataSet; however, style sheets are not easy to write. Alternatively, the style sheet can transform the XML into a displayable format such as HTML. For more information, see How to Apply a Style Sheet Programmatically in a .NET Application in the appendix.
- Parsing XML. To parse XML, you can use the Document Object Model (DOM) or the XmlReader class provided in the Microsoft .NET Framework class library. XmlReader provides fast-forward only, read-only access to XML data, but DOM is more flexible because it provides random read/write access. However, parsing an XML document by using DOM is slower; you must create an instance of XmlDocument (or another XML parser class) and load the entire XML document into memory.
- Sorting XML. You cannot automatically sort XML data. Instead, use one of the following techniques:
  - Deliver the data in presorted order. This option does not support dynamic resorting of data in the calling application.
  - Apply an XSLT style sheet to sort the data dynamically. If necessary, you can alter the sort criteria in the XSLT style sheet at run time, by using DOM.
  - Transform the XML data into a DataSet, and use a DataView object to sort and search the data elements.
- Using private fields. You do not have the option of hiding information.

Representing Business Entities As a Generic DataSet

A generic DataSet is an instance of the DataSet class, which is defined in the System.Data namespace in ADO.NET. A DataSet object contains one or more DataTable objects to represent information that the Data Access Logic Component retrieves from the database.

Figure 7 shows a generic DataSet object for the Product business entity. The DataSet object has a single DataTable to hold information for products. The DataTable has a UniqueConstraint object to denote the ProductID column as a primary key. The DataTable and UniqueConstraint objects are created when the DataSet is created in the Data Access Logic Component.

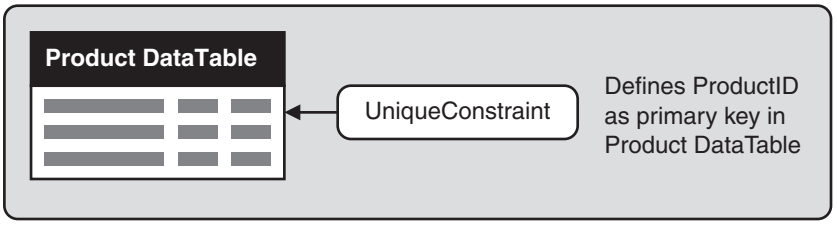

Figure 7
Generic DataSet for the Product business entity

Figure 8 shows a generic DataSet object for the Order business entity. This DataSet object has two DataTable objects to hold information for the orders and order details, respectively. Each DataTable has a corresponding UniqueConstraint object to identify the primary key in that table. Additionally, the DataSet has a Relation object to associate order details with orders.

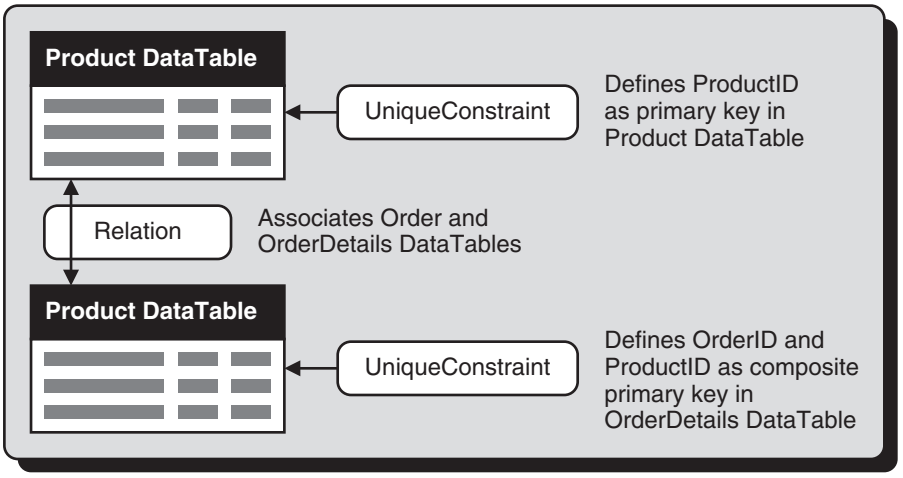

Figure 8
Generic DataSet for the Order business entity

The following code shows how to retrieve a generic DataSet from a Data Access Logic Component, bind the DataSet to a DataGrid control, and then pass the DataSet to the Data Access Logic Component to save data changes:

```
// Create a ProductDALC object
ProductDALC dalcProduct = new ProductDALC();

// Call a method on ProductDALC to get a DataSet containing information for all
// products
DataSet dsProducts = dalcProduct.GetProducts();

// Use DataSet in the client. For example, bind the DataSet to user interface
// controls
dataGrid1.DataSource = dsProducts.Tables[0].DefaultView;
dataGrid1.DataBind();

// When you are ready, pass the updated DataSet to the ProductDALC to save the
// changes back to the database
dalcProduct.UpdateProducts(dsProducts);
```

You can also query and modify the tables, constraints, and relations in a DataSet at run time. For more information, see Creating and Using DataSets.

The advantages of representing business entities as a generic DataSet are as follows:

- Flexibility. DataSets can contain collections of data, and can represent complex data relationships.
- Serialization. DataSets natively support serialization when passing across tiers.
- Data binding. DataSets can be bound to any user interface controls in ASP.NET and Windows Forms applications.
- Sorting and filtering. DataSets can be sorted and filtered by using DataView objects. An application can create several DataView objects for the same DataSet, to view the data in different ways.
- Interchangeability with XML. DataSets can be read or written in XML format. This is a useful technique in remote and disconnected applications, which can receive a DataSet in XML format and recreate the DataSet object locally. Applications can also persist DataSets to XML format while the applications are disconnected from the database.
- Availability of metadata. Full metadata can be provided for a DataSet, in the form of an XSD schema. You can also programmatically obtain metadata for the DataSet by using methods in the DataSet, DataTable, DataColumn, Constraint, and Relation classes.
- Optimistic concurrency. When you are updating data, you can use DataSets, in conjunction with data adapters, to perform optimistic concurrency checks easily.
- Extensibility. If the database schema is modified, the methods in the Data Access Logic Component can create DataSets that contain modified DataTable and DataRelation objects as appropriate. The Data Access Logic Component method signatures do not change. The calling application can be modified to use these new elements in the DataSet.

The disadvantages of representing business entities as a generic DataSet are as follows:

- Client code must access data through collections in the DataSet. To access a table in a DataSet, client code must index into the DataTable collections by using an integer indexer or a string indexer. To access a particular column, you must index into the DataColumn collection by using a column number or a column name. The following example shows how to access the ProductName column for the first row in the Products table:

```
…
// Get the product name for the product in the first row of a DataSet called
// dsProducts. Note the collections are zero-based.
String str = (String)dsProducts.Tables["Products"].Rows[0]["ProductName"];
…
```

Note: Note that there is no compile-time checking of these indexer values. If you specify an invalid table name, column name, or column type, the error is trapped at run time. There is no IntelliSense support when you use generic DataSets.

- High instantiation and marshalling costs. DataSets result in the creation of several subobjects (DataTable, DataRow, and DataColumn), which means that DataSets can take longer to instantiate and marshal than XML strings or custom entity components. The relative performance of DataSets improves as the amount of data increases, because the overhead of creating the internal structure of the DataSet is less significant than the time it takes to populate the DataSet with data.
- Private fields. You do not have the option of hiding information.

Representing Business Entities As a Typed DataSet

A typed DataSet is a class that contains strongly typed methods, properties, and type definitions to expose the data and metadata in a DataSet. For an example of how to create a typed DataSet, see How to Create a Typed DataSet in the appendix.

The following lists describe the advantages and disadvantages of typed DataSets, compared with generic DataSets. Note that typed DataSets exhibit approximately the same instantiation and marshalling performance as generic DataSets.

The advantages of representing business entities as a typed DataSet are as follows:

- Code readability. To access tables and columns in a typed DataSet, you can use typed methods and properties, as shown in the following code:

```
…
// Get the product name for the product in the first row of a typed DataSet
// called dsProducts. Note the collections are zero-based.
String str = dsProducts.Products[0].ProductName;
…
```

In this example, dsProducts is an instance of a typed DataSet. The DataSet has a single DataTable, which is exposed by a property named Products. The columns in this DataTable are exposed by properties such as ProductName, which returns the appropriate data type for the column (rather than just returning Object).

The provision of typed methods and properties makes typed DataSets easier to use than generic DataSets. When you use typed DataSets, IntelliSense is available.

- Compile type checking. Invalid table names and column names are detected at compile time rather than at run time.

The disadvantages of representing business entities as a typed DataSet are as follows:

- Deployment. The assembly containing the typed DataSet class must be deployed to all tiers that use the business entity.
- Support of Enterprise Services (COM+) callers. If a typed DataSet will be used by COM+ clients, the assembly containing the typed DataSet class must be given a strong name and must be registered on client computers. Typically, the assembly is installed in the global assembly cache. These steps will also be required for custom entity classes, as described later in this document.
- Extensibility issues. If the database schema is modified, the typed DataSet class might need to be regenerated to support the new schema. The regeneration process will not preserve any custom code that was implemented in the typed DataSet class. The assembly containing the typed DataSet class must be redeployed to all client applications.
- Instantiation. You cannot instantiate the type by using the new operator.
- Inheritance. Your typed dataset must inherit from DataSet, which precludes the use of any other base classes.

Defining Custom Business Entity Components

Custom classes that represent business entities typically contain the following members:

- Private fields to cache the business entity's data locally. These fields hold a snapshot of the data in the database at the time the data was retrieved from the database by the Data Access Logic Component.
- Public properties to access the state of the entity, and to access subcollections and hierarchies of data inside the entity. The properties can have the same names as the database column names, but this is not an absolute requirement. Choose property names according to the needs of your application, rather than the names in the database.
- Methods and properties to perform localized processing by using the data in the entity component.
- Events to signal changes to the internal state of the entity component.

Figure 9 shows how custom entity classes are used. Note that the entity class has no knowledge of the Data Access Logic Component or the underlying database; all database access is performed by the Data Access Logic Component to centralize data access policies and business logic. Also, the way in which you pass business entity data through the tiers is not directly tied to the format of your business entity representation; for example, you can represent business entities locally as objects, yet choose a different approach (such as scalar values or XML) to pass business entity data to a different tier.

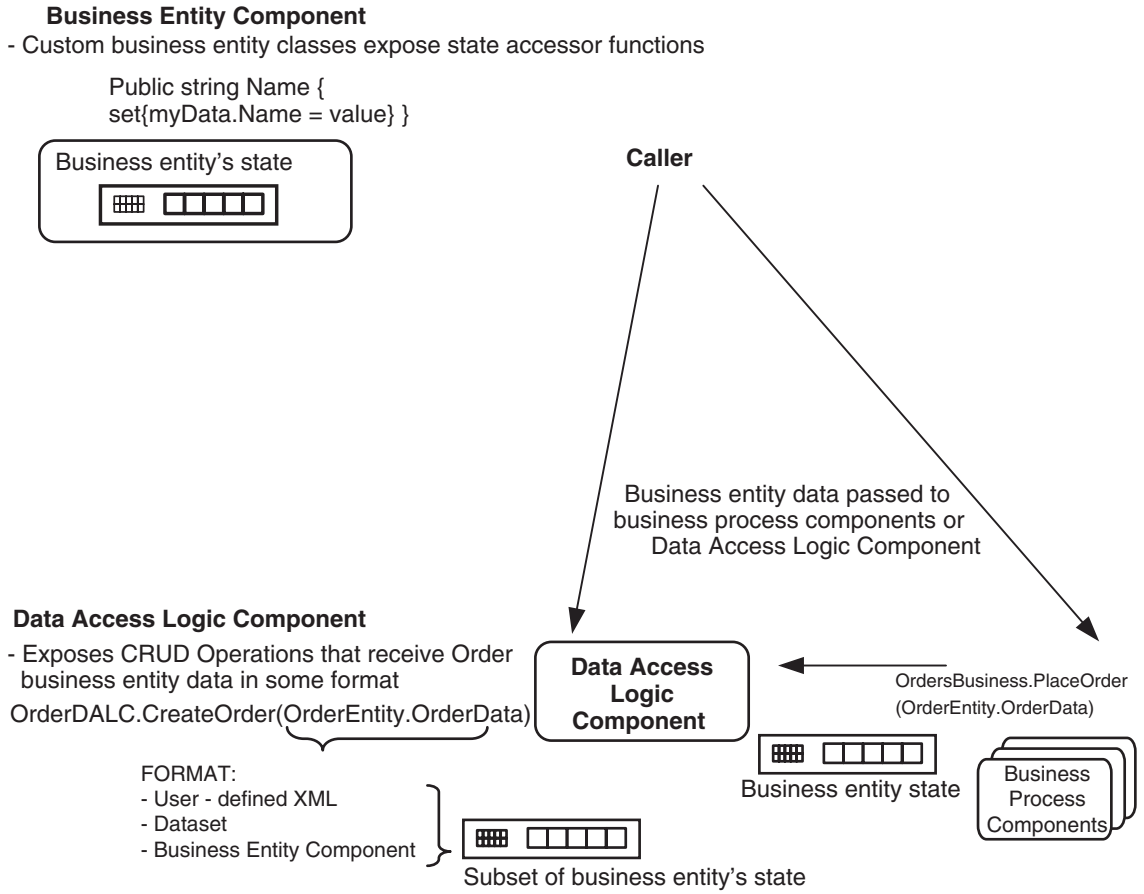

Figure 9

The role of custom business entity components

Recommendations for Defining Custom Business Entity Components

To help you implement your custom entity components, consider the following recommendations:

- Choosing between structs and classes. For simple business entities that do not contain hierarchical data or collections, consider defining a struct to represent the business entity. For complex business entities, or for business entities that require inheritance, define the entity as a class instead. For a comparison of struct and class types, see Structures and Classes.
- Representing the business entity's state. For simple values such as numbers and strings, define fields by using the equivalent .NET data type. For a code example that shows how to define a custom entity, see How to Define a Business Entity Component in the appendix.
- Representing subcollections and hierarchies in a custom Business Entity Component. There are two ways to represent subcollections and hierarchies of data in a custom entity:
  - A .NET collection such as ArrayList. The .NET collection classes offer a convenient programming model for resizable collections, and also provide built-in support for data binding to user interface controls.
  - A DataSet. DataSets are well suited for storing collections and hierarchies of data from a relational database or from an XML document. Additionally, DataSets are preferred if you need to be able to filter, sort, or data bind your subcollections.

    For a code example that shows how to represent collections and hierarchies of data in a custom entity, see How to Represent Collections and Hierarchies of Data in a Business Entity Component in the appendix.
- Supporting data binding for user interface clients. If the custom entity will be consumed by user interfaces and you want to take advantage of automatic data binding, you may need to implement data binding in your custom entity. Consider the following scenarios:
  - Data binding in Windows Forms. You can data bind an entity instance to controls without implementing data binding interfaces in your custom entity. You can also data bind an array or a .NET collection of entities.
  - Data binding in Web Forms. You cannot data bind an entity instance to controls in a Web Form without implementing the IBindingList interface. However, if you want to data bind only sets, you can use an array or a .NET collection without needing to implement the IBindingList interface in your custom entity.

    For a code example that shows how to bind custom entities to user interface controls, see How to Bind Business Entity Components to User Interface Controls in the appendix.

- Exposing events for internal data changes. Exposing events is useful for rich-client user interface design because it enables data to be refreshed wherever it is being displayed. The events should be for internal state only, not for data changes on a server. For a code example that shows how to expose events in a custom entity class, see How to Expose Events in a Business Entity Component in the appendix.
- Making your business entities serializable. Making business entities serializable enables the business entity's state to be persisted in interim states without database interactions. The result can be to ease offline application development and design of complex user interface processes that do not affect business data until they are complete. There are two types of serialization:
  - XML serialization by using the XmlSerializer class. Use XML serialization when you need to serialize only public fields and public read/write properties to XML. Note that if you return business entity data from a Web service, the object is automatically serialized to XML through XML serialization.

    You can perform XML serialization on business entities without implementing any additional code inside the entity. However, only the public fields and public read/write properties in the object are serialized to XML. Private fields, indexers, private properties, read-only properties, and object graphs are not serialized. You can control the resulting XML by using attributes in your custom entity. For more information about how to serialize custom entity components to XML format, see How to Serialize Business Entity Components to XML Format in the appendix.
  - Formatted serialization by using the BinaryFormatter or SoapFormatter class. Use formatted serialization when you need to serialize all the public and private fields and object graphs of an object, or if you will pass an entity component to or from a remoting server.

    The formatter classes serialize all of an object's fields and properties, both public and private. BinaryFormatter serializes the object to binary format, and SoapFormatter serializes an object to SOAP format. Serialization by using BinaryFormatter is faster than serialization by using SoapFormatter. To use either of these formatter classes, you must mark your entity class with the [Serializable] attribute. If you need explicit control over the serialization format, your class must also implement the ISerializable interface. For more information about how to use formatted serialization, see How to Serialize Business Entity Components to Binary Format and How to Serialize Business Entity Components to SOAP Format in the appendix.

Note: Default constructors are not called when an object is deserialized. This constraint is placed on deserialization for performance reasons

The advantages of defining a custom entity are as follows:

- Code readability. To access data in a custom entity class, you can use typed methods and properties, as shown in the following code:

```
// Create a ProductDALC object
ProductDALC dalcProduct = new ProductDALC();

// Use the ProductDALC object to create and populate a ProductEntity object.
// This code assumes the ProductDALC class has a method named GetProduct,
// which takes a Product ID as a parameter (21 in this example) and returns a
// ProductEntity object containing all the data for this product.
ProductEntity aProduct = dalcProduct.GetProduct(21);

// Change the product name for this product
aProduct.ProductName = "Roasted Coffee Beans";
```

  In the preceding example, a product is an instance of a custom entity class named ProductEntity. The ProductDALC class has a method named GetProduct, which creates a ProductEntity object, populates the object with data for a specific product, and returns the ProductEntity object. The calling application can use properties such as ProductName to access the data on the ProductEntity object, and can call methods to manipulate the object.

- Encapsulation. Custom entities can contain methods to encapsulate simple business rules. These methods operate on the business entity data cached in the entity component, rather than accessing the live data in the database. Consider the following example:

```
// Call a method defined in the ProductEntity class.
aProduct.IncreaseUnitPriceBy(1.50);
```

  In the preceding example, the calling application calls a method named IncreaseUnitPriceBy on the ProductEntity object. This change is not made permanent until the ProductEntity object is saved back to the database when the calling application invokes an appropriate method on the ProductDALC object.

- Modeling of complex systems. If you are modeling a complex domain problem that has many interactions between different business entities, it may be beneficial to define custom entity classes to absorb the complexity behind well-defined class interfaces.
- Localized validation. Custom entity classes can perform simple validation tests in their property accessors to detect invalid business entity data. For more information, see How to Validate Data in Property Accessors in Business Entity Component.
- Private fields. You can hide information that you do not want to expose to the caller.

The disadvantages of defining a custom entity are as follows:

- Collections of business entities. A custom entity represents a single business entity, not a collection of business entities. The calling application must create an array or a .NET collection to hold multiple business entities.
- Serialization. You must implement your own serialization mechanism in a custom entity. You can use attributes to control how entity components are serialized, or you can implement the ISerializable interface to control your own serialization.
- Representation of complex relationships and hierarchies in a business entity. You must implement your own mechanism for representing relationships and hierarchies of data in a Business Entity Component. As described previously, DataSets are often the easiest way to achieve this effect.
- Searching and sorting of data. You must define your own mechanism to support searching and sorting of entities. For example, you can implement the IComparable interface to allow entity components to be held in a SortedList or Hashtable collection.
- Deployment. You must deploy, on all physical tiers, the assembly containing the custom entity.
- Support for Enterprise Services (COM+) clients. If a custom entity will be used by COM+ clients, the assembly containing the entity must be given a strong name and must be registered on client computers. Typically, the assembly is installed in the global assembly cache.
- Extensibility issues. If the database schema is modified, you might need to modify the custom entity class and redeploy the assembly.

Defining Custom Business Entity Components with CRUD Behaviors

When you define a custom entity, you can provide methods to completely encapsulate the CRUD operations on the underlying Data Access Logic Component. This is the more traditional object-oriented approach, and may be appropriate for complex object domains. The client application no longer accesses the Data Access Logic Component class directly. Instead, the client application creates an entity component and calls CRUD methods on the entity component. These methods forward to the underlying Data Access Logic Component.

Figure 10 shows the role of custom entity classes with CRUD behaviors.

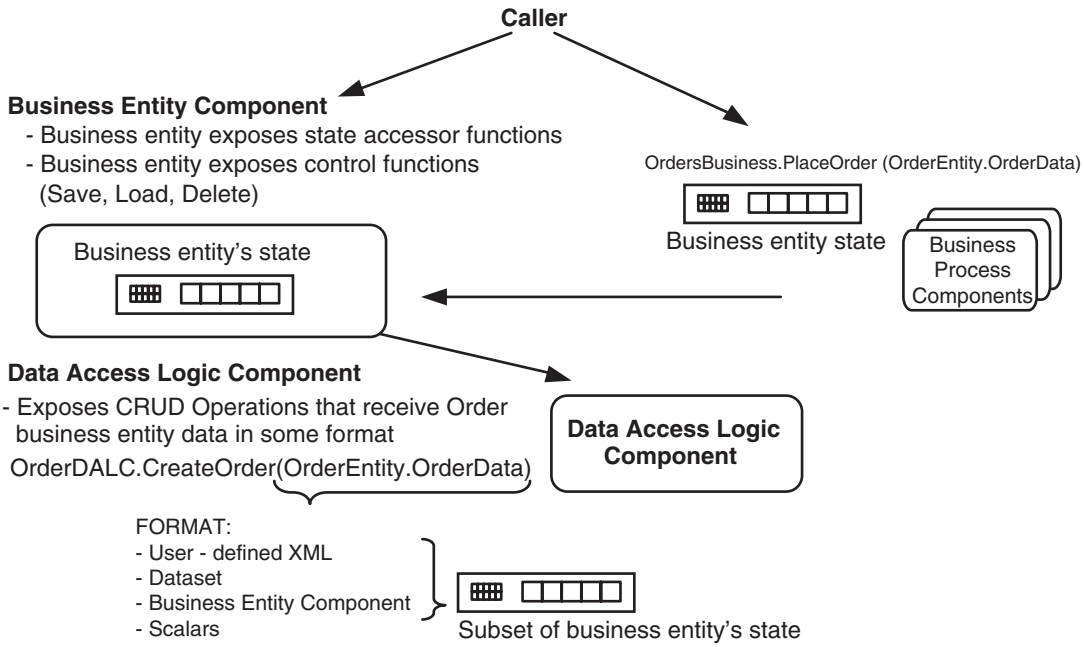

Figure 10

The role of custom business entity components with CRUD behaviors

The advantages of defining custom entity classes with CRUD behaviors are as follows:

- Encapsulation. The custom entity encapsulates the operations defined by the underlying Data Access Logic Component.
- Interface to caller. The caller must deal with only one interface to persist business entity data. There is no need to access the Data Access Logic Component directly.
- Private fields. You can hide information that you do not want to expose to the caller.

The disadvantages of defining custom entity classes with CRUD behaviors are as follows:

- Dealing with sets of business entities. The methods in the custom entity pertain to a single business entity instance. To support sets of business entities, you can define static methods that take or return an array or a collection of entity components.
- Increased development time. The traditional object-oriented approach typically requires more design and development effort than working with existing objects, such as DataSets, requires.

Recommendations for Representing Data and Passing Data Through Tiers

The way in which you represent data throughout your application, and the way in which you pass that data through the tiers, do not necessarily need to be the same. However, having a consistent and limited set of formats yields performance and maintenance benefits that reduce your need for additional translation layers.

The data format that you use should depend on your specific application requirements and how you want to work with the data. There is no universal way to represent your data, especially because many of today's applications are required to support multiple callers. However, it is recommended that you follow these general guidelines:

- If your application mainly works with sets and needs functionality such as sorting, searching, and data binding, Datasets are recommended. However, if your application works with instance data, scalar values will perform better.
- If your application mainly works with instance data, custom business entity components may be the best choice because they prevent the overhead caused when a DataSet represents one row.
- In most cases, design your application to use a data-centric format, such as XML documents or DataSets. You can use the flexibility and native functionality provided by the DataSets to support multiple clients more easily, reduce the amount of custom code, and use a programming API that is familiar to most

developers. Although working with the data in an object-oriented fashion provides some benefits, custom coding complex business entities increases development and maintenance costs in proportion to the amount of features you want to provide.

Transactions

Most of today's applications need to support transactions for maintaining the integrity of a system's data. There are several approaches to transaction management; however, each approach fits into one of two basic programming models:

- Manual transactions. You write code that uses the transaction support features of either ADO.NET or Transact-SQL directly in your component code or stored procedures, respectively.
- Automatic transactions. Using Enterprise Services (COM+), you add declarative attributes to your .NET classes to specify the transactional requirements of your objects at run time. You can use this model to easily configure multiple components to perform work within the same transaction.

This section provides guidance and recommendations to help you implement transaction support in data access logic components and business entity components. For examples and a more in-depth discussion of transactions in .NET, see the .NET Data Access Architecture Guide.

Implementing Transactions

In most circumstances, the root of the transaction is the business process rather than a Data Access Logic Component or a Business Entity Component. The reason is that business processes typically require transactions that span multiple business entities, not just a single business entity.

However, situations may arise where you need to perform transactional operations on a single business entity without the assistance of a higher-level business process. For example, to add a new customer to the database discussed earlier, you must perform the following operations:

- Insert a new row in the Customer table.
- Insert a new row or rows in the Address table.

Both of these operations must succeed, or the customer will not be added to the database. If the Customer business entity will never be a part of a larger business process that will initiate the transaction, use manual transactions within the Customer business entity. Manual transactions are significantly faster than automatic transactions because they do not require any interprocess communication with the Microsoft Distributed Transaction Coordinator (DTC).

Figure 11 shows how to decide whether to use manual transactions or automatic transactions. Due to the overhead of COM+ transactions, the recommended approach is to push the transaction to the database and control the transactional behavior in stored procedures where possible.

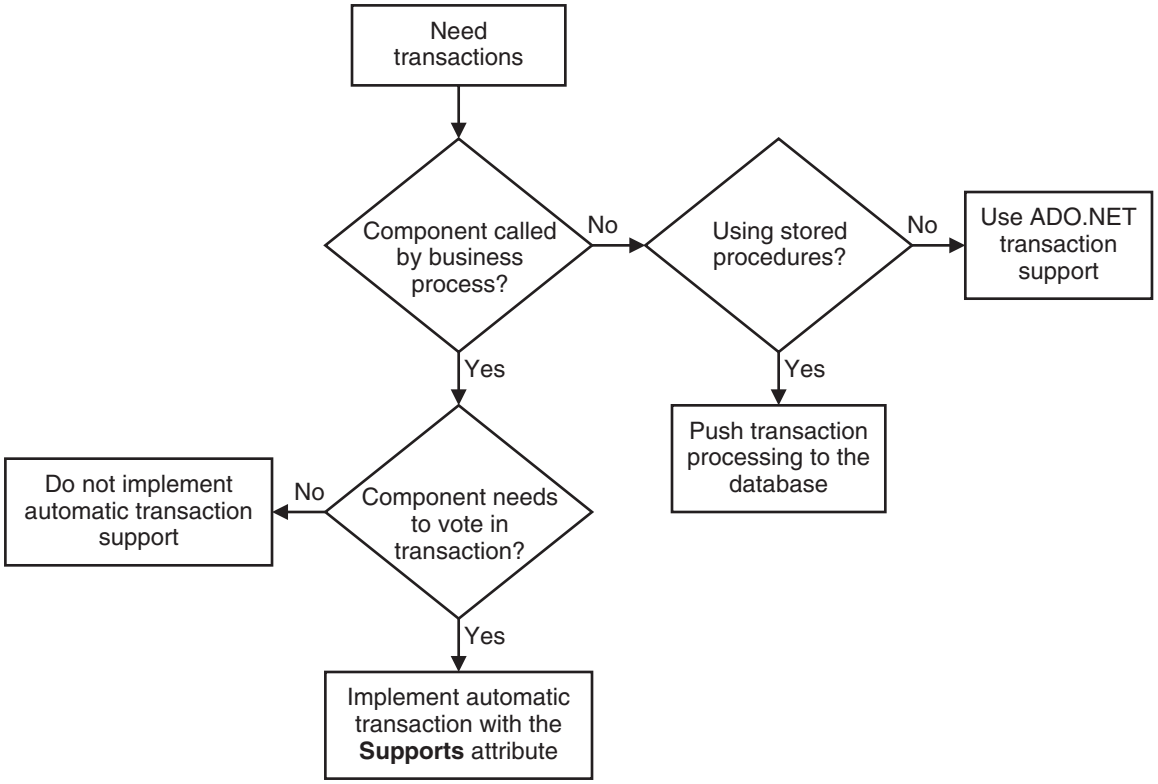

Figure 11
Deciding how to implement transactions

Note: If you are calling from an ASP.NET-based client, and there is no business process to initiate the transaction, you may be tempted to start the transaction in the ASP.NET code-behind. This is not a good design; you should never initiate a transaction from an ASP.NET-based client. Instead, separate the presentation of the data from the business process. Performance is also an issue due to issues such as network latency, because this is the most common layer to be physically deployed on another tier.

Recommendations for Using Manual Transactions in Data Access Logic Components

When you implement manual transactions in data access logic components, consider the following recommendations:

- Where possible, perform your processing in stored procedures. Use the Transact-SQL statements BEGIN TRANSACTION, END TRANSACTION, and ROLLBACK TRANSACTION to control transactions. For a code example, see "How To Perform Transactions With Transact-SQL" in the .NET Data Access Architecture Guide.
- If you are not using stored procedures, and the data access logic components will not be called from a business process, you can control transactions programmatically by using ADO.NET. For a code example, see "How to Code ADO.NET Manual Transactions" in the .NET Data Access Architecture Guide.

Recommendations for Using Automatic Transactions in Data Access Logic Components

Despite the overhead associated with COM+ transactions, automatic transactions provide a simpler programming model than manual transactions, and are necessary when your transactions span multiple distributed data sources as they work in conjunction with the DTC. If you implement automatic transactions in data access logic components, consider the following recommendations:

- The Data Access Logic Component must inherit from the ServicedComponent class in the System.EnterpriseServices namespace. Note that any assembly registered with COM+ services must have a strong name. For more information about strong-named assemblies, see Creating and Using Strong-Named Assemblies.
- Annotate the Data Access Logic Component with the Transaction(TransactionOption.Supported) attribute so that you can perform read and write operations in the same component. This option avoids the overhead of transactions where they are not required—unlike Transaction(TransactionOption.Required), which always requires a transaction.

The following code sample shows how to support automatic transactions in a Data Access Logic Component class:

```
using System.EnterpriseServices;

[Transaction(TransactionOption.Supported)]
public class CustomerDALC : ServicedComponent
{
  ...
}
```

If you use automatic transactions, your data access logic components should vote in transactions to indicate whether the operation succeeded or failed. To vote implicitly, annotate your methods by using the AutoComplete attribute and throw an exception if the operation fails. To vote explicitly, call the SetComplete or SetAbort method on the ContextUtil class.

For more information about automatic transactions, see "Using Automatic Transactions" in the .NET Data Access Architecture Guide.

Using Automatic Transactions in Business Entity Components

If you implement custom business entity components that have behaviors, you can use automatic transactions to specify the transactional behavior of these objects. The recommendations for using automatic transactions to specify the transactional behavior of business entity components are the same as the previously listed recommendations for implementing automatic transactions in data access logic components.

Note: If the Business Entity Component does not contain any business logic that requires it to vote in the transaction, it can ignore the transaction context altogether. The custom Business Entity Component does not need to inherit from ServicedComponent; the transaction context will still flow, but the entity component will ignore the context.

Validations

You can perform data validation at many tiers in your application. Different types of validation are appropriate in each tier:

- The client application can validate business entity data locally, before the data is submitted.
- Business processes can validate business documents as the documents are received by using an XSD schema.
- Data access logic components and stored procedures can validate data to ensure referential integrity and to enforce constraints and nontrivial business rules.

There are two general kinds of validation:

- Point-in-time validation. This is validation that is performed at a specific point in time. For example, a business process validates an XML document when the document is received.

- Continuous validation. This is validation that is performed on an ongoing basis at many different levels in your application. Examples of continuous validation include the following:
  - User interfaces can specify maximum field lengths to prevent the user from entering strings that are too long.
  - DataSets can specify the maximum length of data columns.
  - Custom business entity components can perform range checks, length checks, non-null checks, and other simple tests on entity data.
  - Data access logic components, stored procedures, and the database itself can perform similar tests to ensure that data is valid before it is saved in the database.

At times, you may want to implement an out-of-band aggregator or a transformer process. This approach may be useful if the validations and transformations change often, but it incurs a performance penalty. For example, if an ISV wanted to use the same components to support two versions of a database schema, you could create a separate component to perform validation and transformations between the two database schema versions.

How to Validate XML by Using an XSD Schema

To validate an XML document against an XSD schema, take the following steps:

1. Create an XmlValidatingReader object as a wrapper around an XmlTextReader object, as shown in the following code:

```
' Create an XmlValidatingReader object, to read and validate Product.xml
XmlTextReader tr = new XmlTextReader("Product.xml");
XmlValidatingReader vr = new XmlValidatingReader(tr);
```

2. Specify the type of validation required, by using the ValidationType enumeration. The .NET Framework supports three types of XML validation:
   - Document type definitions (DTDs); specify ValidationType.DTD.
   - Microsoft XML data-reduced (XDR) schemas; specify ValidationType.XDR.
   - W3C-standard XSD schemas; specify ValidationType.Schema.

   The following code shows the use of the ValidationType enumeration:

```
vr.ValidationType = ValidationType.Schema;  ' Specify XSD schema validation
```

3. Register a validation event-handler method, as shown in the following code:

```
vr.ValidationEventHandler += new ValidationEventHandler(MyHandlerMethod);
```

4. Provide an implementation for the validation event-handler method, as shown in the following code:

```
public void MyHandlerMethod(object sender, ValidationEventArgs e)
{
   Console.WriteLine("Validation Error: " + e.Message);
}
```

5. Read and validate the document, as shown in the following code. Validation errors are picked up by the validation event-handler method.

```
try
{
   while (vr.Read())
   {
      // Process the XML data, as appropriate ...
   }
}
catch (XmlException ex)
{
   Console.WriteLine("XmlException: " + ex.Message);
}
vr.Close();
```

How to Validate Data in Property Accessors in Business Entity Components

The following code fragment shows how to perform simple validation in property accessors in a custom entity. If the validation test fails, you can throw an exception to indicate the nature of the problem. You can also use regular expressions within your set property accessors to validate specific data and formats.

```
public class ProductDALC
{
 …
  public short ReorderLevel
  {
    get { return reorderLevel; }
  }
  set
  {
    if (value < 0)
    {
      throw new ArgumentOutOfRangeException("ReorderLevel cannot be negative.");
    }
    reorderLevel = value;
  }

  // Plus other members in the ProductDALC class...
}
```

Exception Management

When errors occur in .NET applications, the general advice is to throw exceptions rather than return error values from your methods. This advice has implications for the way you write data access logic components and business entity components. There are two general kinds of exceptions that will occur:

- Technical exceptions, which include:
  - ADO.NET
  - Connection to database
  - Resources (such as database, network share, and Message Queuing) are unavailable
- Business logic exceptions, which include:
  - Validation errors
  - Errors in stored procedures that implement business logic

Recommendations for Managing Exceptions in a Data Access Logic Component

Data access logic components should propagate exceptions, and should wrap exception types only if doing so makes the exception easier for the client to manage. Wrapping the exceptions in two main exception types (technical and business) helps exception handling structure and exception publishing logic for the potentially diverse callers.

Your application should publish exception information. Technical exceptions can be published to a log that is monitored by administrators or by a Windows Management Instrumentation (WMI) monitoring tool like Microsoft Operations Manager. Business exceptions can be published to an application-specific log. In general, allow the exceptions to propagate from your Data Access Logic Component and be published by the caller so that you have the entire context of the exception.

The following example illustrates these recommendations:

```
public class CustomerDALC
{
  public void UpdateCustomer(Dataset aCustomer)
  {
    try
    {
      // Update the customer in the database...
    }
    catch (SqlException se)
    {
```

```
      // Catch the exception and wrap, and rethrow
      throw new DataAccessException("Database is unavailable", se);
    }
    finally
    {
      // Cleanup code
    }
  }
}
```

Recommendations for Managing Exceptions in Business Entity Components

Business entity components should propagate exceptions to callers. Business entity components may also raise exceptions if they are performing validation, or if the caller tries to perform an operation without providing all the required data for the operation.

The following example illustrates how a business entity component can raise an exception. In this example, the Update method throws an exception if the customer's first name is not provided:

```
public class CustomerEntity
{
  public void Update()
  {
   // Check that the user has provided the required data. In this case a first
   // name for the customer
    if (FirstName == "" )
    {
       // Throw a new application exception that you have defined
       throw new MyArgumentException("You must provide a First Name.);
    }
    ...
  }
}
```

For more information about dealing with exceptions in .NET applications, see Exception Management in .NET. You can inherit your custom technical exceptions and custom business exceptions from the ApplicationException class or the BaseApplicationException class provided in the Exception Management Application Block.

Authorization and Security

This section explains how security applies to your data access logic components and your business entity components. The .NET common language runtime uses permissions objects to implement its mechanism for enforcing restrictions on managed code. There are three kinds of permissions objects, each with a specific purpose:

- Code access security. These permissions objects are used to protect resources and operations from unauthorized use.
- Identity. These permissions objects specify the required identity characteristics that an assembly must have in order to run.
- Role-based security. These permissions objects provide a mechanism for discovering whether a user (or the agent acting on the user's behalf) has a particular identity or is a member of a specified role. The PrincipalPermission object is the only role-based security permissions object.

Managed code can discover the identity or the role of a principal by using a Principal object, which contains a reference to an Identity object. It might be helpful to compare identity and principal objects to familiar concepts like user and group accounts. In the .NET Framework, identity objects represent users, whereas roles represent memberships and security contexts. The principal object encapsulates both an identity object and a role. Applications in the .NET Framework grant rights to the principal object based on its identity or, more commonly, its role membership.

For more information about permissions and security in .NET, see Key Security Concepts.

Recommendations for Security in Data Access Logic Components

Data access logic components are designed to be used by other application components, and are the last place in your application code where you can implement security before the caller has access to your data.

Often, data access logic components can rely on the security context set by the caller. However, there are some situations where the Data Access Logic Component should perform its own authorization checks to determine whether a principal is allowed to perform a requested action. Authorization occurs after authentication and uses information about the principal's identity and roles to determine what resources the principal can access.

Perform authorization checks at the Data Access Logic Component level if you need to:

- Share data access logic components with developers of business processes that you do not fully trust.
- Protect access to powerful functions exposed by the data stores.

After you define identity and principal objects, there are three different ways to perform role-based security checks:

- Use the PrincipalPermission object to perform imperative security checks.
- Use the PrincipalPermissionAttribute attribute to perform declarative security checks.
- Use the properties and the IsInRole method in the Principal object to perform explicit security checks.

The following code sample shows how to use PrincipalPermissionAttribute to specify a declarative role-based security check on a method in a Data Access Logic Component class:

```
using System;
using System.Security.Permissions;

public class CustomerDALC
{

  public CustomerDALC()
  {
  }

  // Use PrincipalPermissionAttribute to demand that the caller of this method
  // has an identity named "MyUser", and belongs to the "Administrator" role.
  [PrincipalPermissionAttribute(SecurityAction.Demand,
                                Name="MyUser", Role="Administrator")]
  public void DeleteCustomer(string customerID)
  {
    // Delete customer code here
  }
}
```

The following code shows how to create a principal object that has the required identity and role, so that you can call the DeleteCustomer method on a CustomerDALC object:

```
using System;
using System.Security.Principal;
using System.Threading;

public class MainClass
{
  public static int Main(string[] args)
  {
    Console.Write("User Name: ");
    string UserName = Console.ReadLine();

    Console.Write("Password: ");
    string Password = Console.ReadLine();
```

```
    if (Password == "password" && UserName == "MyUser")
    {
      // Create a generic identity with the name "MyUser"
      GenericIdentity MyIdentity = new GenericIdentity("MyUser");

      // Create roles
      String[] MyString = {"Administrator", "User"};

      // Create a generic principal
      GenericPrincipal MyPrincipal = new GenericPrincipal(MyIdentity, MyString);

      // Set this thread's current principal, for use in role-based security
      Thread.CurrentPrincipal = MyPrincipal;
    }

    // Create a CustomerDALC object, and try to call its DeleteCustomer method.
    // This will only succeed if the current principal's identity and role are OK.
    CustomerDALC c = new CustomerDALC();
    c.DeleteCustomer("VINET");
  }
}
```

Windows Authentication

Ideally, you should use Windows Authentication, rather than using SQL Server Authentication, when you connect to the database. However, you should use service accounts and avoid impersonating through to the database, because this will impede connection pooling. Connection pooling requires identical connection strings; if you try to open the database by using different connection strings, you will create separate connection pools, which will limit scalability.

For more information about Windows Authentication and connection pooling, see the "Managing Database Connections" section in the .NET Data Access Architecture Guide.

Secure Communication Recommendations

To achieve secure communication between calling applications and data access logic components, consider the following recommendations:

- If your data access logic components are called over the wire from diverse tiers, and the exchange involves sensitive information that needs to be protected, use Distributed Component Object Model (DCOM), Secure Sockets Layer (SSL), or Secure Internet Protocol (IPSec) secure communication technologies.
- If data is stored encrypted in a database, data access logic components are usually responsible for encrypting and decrypting the data. If the risk of exposing the information is high, strongly consider securing the communication channel to and from the data access logic components.

Recommendations for Security in Business Entity Components

If you implement your business entities as data structures (such as XML or DataSets), you do not need to implement security checks. However, if you implement your business entities as custom business entity components with CRUD operations, consider the following recommendations:

- If the entities are exposed to business processes that you do not fully trust, implement authorization checks in the business entity components and in the data access logic components. If you do implement checks at both levels, however, you may encounter the maintenance issue of keeping the security policies synchronized.
- Business entity components should not deal with communication security or data encryption. Leave these tasks to the corresponding Data Access Logic Component.

Deployment

This section provides recommendations to help you decide how to deploy data access logic components and business entity components.

Deploying Data Access Logic Components

There are two ways to deploy data access logic components:

- Deploy data access logic components together with the business process objects. This deployment method provides optimum performance for data transfers, and has several additional technical benefits:
  - Transactions can flow seamlessly between the business process objects and the data access logic components. However, transactions do not flow seamlessly across remoting channels. In the remoting scenario, you need to implement transactions by using DCOM. Furthermore, if the business process and the Data Access Logic Component were separated by a firewall, you would require firewall ports open between both physical tiers to enable DTC communication.
  - Deploying business process objects and data access logic components together reduces the number of transaction failure nodes.
  - The security context flows automatically between the business process objects and the data access logic components. There is no need to set principal objects.
- Deploy data access logic components together with the user-interface code. Data access logic components are sometimes used directly from UI components and UI process components. To increase performance in Web scenarios, you can

deploy data access logic components together with the UI code; this deployment method enables the UI layer to take advantage of data reader streaming for optimum performance. However, if you do consider this deployment method, bear in mind the following:

- A common reason for not deploying data access logic components together with UI code is to prevent direct network access to your data sources from your Web farms.
- If your Web farm is deployed in a DMZ scenario, firewall ports must be opened to access SQL Server. If you are using COM+ transactions, additional firewall ports must be opened for DTC communication. For more information, see the .NET Data Access Architecture Guide.

Deploying Business Entities

Business entities are used at many different tiers in your application. Depending on how you implement your business entities, you may need to deploy them to multiple locations if your application spans physical tiers. The following list describes how to deploy business entities in different implementation scenarios:

- Deploying Business Entities implemented as typed DataSets. The typed DataSet class must be accessed by the Data Access Logic Component and by the calling application. Therefore, the recommendation is to define typed DataSet classes in a common assembly to be deployed on multiple tiers.
- Deploying Business Entities implemented as custom business entity components. The custom entity class may need to be accessed by the Data Access Logic Component, depending on how you defined the method signatures in the Data Access Logic Component. Follow the same recommendation as for typed DataSets by defining custom entity classes in a common assembly to be deployed on multiple tiers.
- Deploying Business Entities implemented as generic DataSets or XML strings. Generic DataSets and XML strings do not represent a separate data type. There are no deployment issues for business entities implemented in these formats.

Appendix

Appendix Contents

How to Define a Data Access Logic Component Class

How to Use XML to Represent Collections and Hierarchies of Data

How to Apply a Style Sheet Programmatically in a .NET Application

How to Create a Typed DataSet

How to Define a Business Entity Component

How to Represent Collections and Hierarchies of Data in a Business Entity Component

How to Bind Business Entity Components to User Interface Controls

How to Expose Events in a Business Entity Component

How to Serialize Business Entity Components to XML Format

How to Serialize Business Entity Components to SOAP Format

How to Serialize Business Entity Components to Binary Format

How to Define a Data Access Logic Component Class

The code that follows is a sample definition of a class named CustomerDALC, which is the Data Access Logic Component class for Customer business entities. The CustomerDALC class implements the CRUD operations for the Customer business entity and provides additional methods to encapsulate business logic for this object.

```
public class CustomerDALC
{
  private string conn_string;

  public CustomerDALC()
  {
    // Acquire the connection string from a secure or encrypted location
    // and assign to conn_string
  }

  public CustomerDataSet GetCustomer(string id)
  {
    // Code to retrieve a typed DataSet containing Customer data
  }

  public string CreateCustomer(string name,
                               string address, string city, string state, string
zip)
  {
```

```
    // Code to create a new customer in the database, based on the scalar
    // parameters passed into this method.
    // Return the customerID from this method.
  }

  public void UpdateCustomer(CustomerDataSet updatedCustomer)
  {
    // Code to update the database, based on the Customer data sent in as a
    // parameter of type CustomerDataSet
  }

  public void DeleteCustomer(string id)
  {
    // Code to delete the customer with the specified ID
  }

  public DataSet GetCustomersWhoPurchasedProduct(int productID)
  {
    // Code to retrieve customers using a generic DataSet, because this method is
    // not retrieving all the information associated with a customer
  }
}
```

How to Use XML to Represent Collections and Hierarchies of Data

The following example shows how to represent collections and hierarchies of data in an XML document. The XML document represents a single order made by a customer; note that the <OrderDetails> element holds a collection of order detail information for the order.

```
<Order xmlns="urn:aUniqueNamespace">
  <OrderID>10248</OrderID>
  <CustomerID>VINET</CustomerID>
  <OrderDate>1996-07-04</OrderDate>
  <ShippedDate>1996-07-16</ShippedDate>
  <OrderDetails>
    <OrderDetail>
      <ProductID>11</ProductID>
      <UnitPrice>14.00</UnitPrice>
      <Quantity>12</Quantity>
    </OrderDetail>
    <OrderDetail>
      <ProductID>42</ProductID>
      <UnitPrice>9.80</UnitPrice>
      <Quantity>10</Quantity>
    </OrderDetail>
    <OrderDetail>
      <ProductID>72</ProductID>
      <UnitPrice>34.80</UnitPrice>
      <Quantity>5</Quantity>
    </OrderDetail>
  </OrderDetails>
</Order>
```

How to Apply a Style Sheet Programmatically in a .NET Application

To apply a style sheet programmatically in a .NET application, take the following steps:

1. Import the System.Xml.Xsl namespace as shown in the following code. The System.Xml.Xsl namespace contains the XSLT transformation classes in the .NET Framework class library.

```
using System.Xml.Xsl;
```

2. Create an XslTransform object as shown in the following code:

```
XslTransform stylesheet = new XslTransform();
```

3. Load the required style sheet into the XslTransform object as shown in the following code:

```
stylesheet.Load("MyStylesheet.xsl");
```

4. Call the Transform method on the XslTransform object as shown in the following code. Calling the Transform method specifies the names of the XML source document and the result document.

```
stylesheet.Transform(sourceDoc, resultDoc);
```

How to Create a Typed DataSet

You can use typed DataSets to represent business entities. There are several ways to create a typed DataSet:

- From a data adapter within Visual Studio .NET
- From an XSD schema file within Visual Studio .NET
- From the .NET Framework command prompt window by using the XSD Schema Definition Tool (xsd.exe)

Note: It is also possible to define a typed DataSet programmatically by inheriting from the DataSet and defining methods, properties, and nested classes to represent the structure of the DataSet. The easiest way to do this is to create a typed DataSet by using one of the procedures that follow, and then use this typed DataSet class as the basis for your own typed DataSet classes in the future.

Creating a Typed DataSet by Using a Data Adapter

To create a typed DataSet by using a data adapter, follow these steps:

1. In Visual Studio .NET, add a data adapter to your form or component. In the data adapter Configuration Wizard, specify connection information for the data adapter. Also specify SQL strings or stored procedures for the Select, Insert, Update, and Delete commands for your data adapter, as appropriate.
2. In the Component Designer, right-click the Data Adapter object, and then click Generate DataSet.
3. In the Generate DataSet dialog box, click New, type a name for the new DataSet class, and then click OK.
4. To verify that the typed DataSet has been created, in Solution Explorer, click the Show All Files button. Expand the node for the XSD schema file, and verify that there is a code file associated with the XSD schema. The code file defines the new typed DataSet class.

Creating a Typed DataSet from an XSD Schema File

To create a typed DataSet from an XSD schema file by using Visual Studio .NET, follow these steps:

1. In Visual Studio .NET, create a new project or open an existing project.
2. Add an existing XSD schema to the project, or create a new XSD schema in the Component Designer.
3. In Solution Explorer, double-click the XSD schema file to view the XSD schema in the Component Designer.
4. Select the primary XSD schema element in the Component Designer.
5. On the Schema menu, click Generate DataSet.
6. To verify that the typed DataSet has been created, in Solution Explorer, click the Show All Files button. Expand the node for the XSD schema file, and verify that there is a code file associated with the XSD schema. The code file defines the new typed DataSet class.

Creating a Typed DataSet by Using the XSD Schema Definition Tool (xsd.exe)

The XML Schema Definition tool can generate a typed DataSet from an XSD schema file, an XDR schema file, or an XML instance document. The following command uses an XSD schema file named XsdSchemaFile.xsd to generate a typed DataSet in a Visual C# source file named XsdSchemaFile.cs in the current directory:

```
xsd /dataset /language:C# XsdSchemaFile.xsd
```

For more information, see Generating a Strongly Typed DataSet.

How to Define a Business Entity Component

The following example shows how to define a custom entity class for the Product business entity:

```
public class ProductEntity
{
  // Private fields, to hold the state of the Product entity
  private int productID;
  private string productName;
  private string quantityPerUnit;
  private decimal unitPrice;
  private short unitsInStock;
  private short unitsOnOrder;
  private short reorderLevel;

  // Public properties, to expose the state of the entity
  public int ProductID
  {
    get { return productID; }
    set { productID = value; }
  }
  public string ProductName
  {
    get { return productName; }
    set { productName = value; }
  }
  public string QuantityPerUnit
  {
    get { return quantityPerUnit; }
    set { quantityPerUnit = value; }
  }
  public decimal UnitPrice
  {
    get { return unitPrice; }
    set { unitPrice = value; }
  }
  public short UnitsInStock
  {
    get { return unitsInStock; }
    set { unitsInStock = value; }
  }
  public short UnitsOnOrder
  {
    get { return unitsOnOrder; }
    set { unitsOnOrder = value; }
  }
  public short ReorderLevel
  {
    get { return reorderLevel; }
    set { reorderLevel = value; }
  }
```

```
  // Methods and properties to perform localized processing
  public void IncreaseUnitPriceBy(decimal amount)
  {
    unitPrice += amount;
  }
  public short UnitsAboveReorderLevel
  {
    get { return (short)(unitsInStock - reorderLevel); }
  }
  public string StockStatus
  {
    get
    {
      return "In stock: " + unitsInStock + ", on order: " + unitsOnOrder;
    }
  }
}
```

How to Represent Collections and Hierarchies of Data in a Business Entity Component

The following example shows how to define a custom entity class for the Order business entity. Each order comprises many order items; these order items are stored in a DataSet in the OrderEntity class.

```
public class OrderEntity
{
  // Private fields, to hold the order information
  private int orderID;
  private string customerID;
  private DateTime orderDate;
  private DateTime shippedDate;

  // Private field, to hold the order details information
  private DataSet orderDetails;

  // Public properties, to expose the order information
  public int OrderID
  {
    get { return orderID; }
    set { orderID = value; }
  }
  public string CustomerID
  {
    get { return customerID; }
    set { customerID = value; }
  }
  public DateTime OrderDate
  {
    get { return orderDate; }
    set { orderDate = value; }
```

```
  }
  public DateTime ShippedDate
  {
    get { return shippedDate; }
    set { shippedDate = value; }
  }

  // Public property, to expose the order details information
  public DataSet OrderDetails
  {
    get { return orderDetails; }
    set { orderDetails = value; }
  }

  // Additional method, to simplify access to order details information
  public bool IsProductOrdered(int productID)
  {
    // Primary key column must be defined in DataTable
    DataRow row = orderDetails.Tables[0].Rows.Find(productID);

    if (row != null)
       return true;
    else
       return false;
  }

  // Additional property, to simplify access to order details information
  public int NumberOfOrderItems
  {
    get
    {
      return orderDetails.Tables[0].Rows.Count;
    }
  }
}
```

Note the following points concerning the OrderEntity class:

- The class contains private fields to hold information about the order. There is also a private DataSet field, to hold additional details about the order. The Data Access Logic Component will populate all these fields when it creates the OrderEntity object.
- The class has public properties to expose information about the order. There is also a property to expose the DataSet, to enable the calling application to access the order details.
- The class has an additional method and property to provide simplified access to order details:
  - The IsProductOrdered method receives a ProductID parameter and returns a Boolean value to indicate whether this product appears in the order.
  - The NumberOfOrderItems property indicates the number of order lines in the order.

How to Bind Business Entity Components to User-Interface Controls

You can bind user interface controls to custom entities in Windows Forms and in ASP.NET applications. There are two possible scenarios:

- Binding a single business entity to user interface controls. The following code sample shows how you can get an OrderEntity object from the OrderDALC object and bind the OrderEntity object to controls in a Windows Form. When the user changes values in these controls, the data in the underlying OrderEntity object is also changed automatically.

```
// Create an OrderDALC object.
OrderDALC dalcOrder = new OrderDALC();

// Use dalcOrder to get an OrderEntity object for order ID 10248.
// This code assumes the OrderDALC class has a method named GetOrder(), which
// returns an OrderEntity object for a particular order ID.
OrderEntity order = dalcOrder.GetOrder(10248);

// Bind the OrderID property of the OrderEntity to a TextBox control.
textBox1.DataBindings.Add("Text", order, "OrderID");

// Bind the CustomerID property of the OrderEntity to another TextBox control.
textBox2.DataBindings.Add("Text", order, "CustomerID");

// Bind the OrderDate property of the OrderEntity to a DatePicker control.
dateTimePicker1.DataBindings.Add("Value", order, "OrderDate");

// Bind the ShippedDate property of the OrderEntity to another DatePicker
// control.
dateTimePicker2.DataBindings.Add("Value", order, "ShippedDate");

// Bind the OrderDetails DataSet of the OrderEntity to a DataGrid control.
// The DataGrid displays each DataRow of the DataSet in a separate row in the
// grid.
dataGrid1.DataSource = order.OrderDetails.Tables[0].DefaultView;
```

When you are ready, you can pass the modified OrderEntity object back to the OrderDALC so that the data can be saved in the database, as shown in the following code.

```
// Save the OrderEntity object back to the database, via dalcOrder.
// This code assumes the OrderDALC class has a method named UpdateOrder(),
which
// receives an OrderEntity parameter and updates the appropriate entry in the
// database
dalcOrder.UpdateOrder(order);
```

- Binding a collection of business entities to a DataGrid control. The following code sample shows how to get an array of OrderEntity objects from the OrderDALC and then bind the array to a DataGrid control in a Windows Form. The DataGrid displays each array element (that is, each OrderEntity object) in a separate row in the grid.

```
// Create an OrderDALC object.
OrderDALC dalcOrder = new OrderDALC();

// Use dalcOrder to get an array of OrderEntity objects for customer "VINET".
// This code assumes the OrderDALC class has a method named
GetOrdersForCustomer(),
// which returns an array of OrderEntity objects for a particular customer.
OrderEntity[] orderEntities = dalcOrder.GetOrdersForCustomer("VINET");

// Bind the array to a DataGrid control.
dataGrid1.DataSource = orderEntities;
```

  When you are ready, you can pass the modified array back to the OrderDALC so that the data can be saved in the database, as shown in the following code:

```
// Save the OrderEntity objects back to the database, via dalcOrder.
// This code assumes OrderDALC has a method named UpdateOrders(), which takes an
// array
// of OrderEntity objects and updates the appropriate entries in the database.
dalcOrder.UpdateOrders(orderEntities);
```

How to Expose Events in a Business Entity Component

Custom entities can raise events when the business entity state is modified. These events are useful for rich client, user-interface design because data can be refreshed wherever it is being displayed. The following code sample shows how to raise business entity-related events in the OrderEntity class:

```
// Define a common event class for all business entity events
public class EntityEventArgs : EventArgs
{
  // Define event members, to provide information about the event
}

// Define a delegate specifying the signature for business entity-related events
public delegate void EntityEventHandler(Object source, EntityEventArgs e);

// Define a custom Entity class that raises events when the business entity state
// changes
public class OrderEntity
{
```

```
  // Define 'before' and 'after' events for business entity state changes
  public event EntityEventHandler BeforeChange, AfterChange;

  // Private fields, to hold the business entity's state
  private int orderID;
  private int customerID;
  private DateTime orderDate;
  private DateTime shippedDate;
  private DataSet orderDetails;

  // Public properties, to expose the business entity's state
  public int OrderID
  {
    get { return orderID; }
    set
    {
      BeforeChange(this, new EntityEventArgs());   // Raise a 'before' event
      orderID = value;
      AfterChange(this, new EntityEventArgs());    // Raise an 'after' event
    }
  }
  public int CustomerID
  {
    get { return customerID; }
    set
    {
      BeforeChange(this, new EntityEventArgs());   // Raise a 'before' event
      customerID = value;
      AfterChange(this, new EntityEventArgs());    // Raise an 'after' event
    }
  }
  public DateTime OrderDate
  {
    get { return orderDate; }
    set
    {
      BeforeChange(this, new EntityEventArgs());   // Raise a 'before' event
      orderDate = value;
      AfterChange(this, new EntityEventArgs());    // Raise an 'after' event
    }
  }
  public DateTime ShippedDate
  {
    get { return shippedDate; }
    set
    {
      BeforeChange(this, new EntityEventArgs());   // Raise a 'before' event
      shippedDate = value;
      AfterChange(this, new EntityEventArgs());    // Raise an 'after' event
    }
  }

  // Additional members, as required ...
}
```

Note the following points concerning the preceding code:

- The EntityEvent class provides information about business entity-related events. The EntityEventHandler delegate specifies the signature for all business entity-related events raised by custom entity classes. The delegate signature follows the recommended .NET Framework guidelines for event-handler delegates. For guidelines on how to define and use events in .NET, see Event Usage Guidelines.
- The OrderEntity class defines two events named BeforeChange and AfterChange.
- The property setters in OrderEntity raise a BeforeChange event before the business entity state is changed, and an AfterChange event after the business entity state is changed.

How to Serialize Business Entity Components to XML Format

This section discusses the following issues:

- Using XmlSerializer to serialize a custom entity object
- XML serialization of objects in XML Web services
- Default XML format of a serialized custom entity object
- Controlling the XML format of a serialized custom entity object

Using XmlSerializer to Serialize a Custom Entity Object

The following code sample shows how to use the XmlSerializer class to serialize an OrderEntity object to XML format:

```
using System.Xml.Serialization;     // This namespace contains the XmlSerializer
class
...
// Create an XmlSerializer object, to serialize OrderEntity-type objects
XmlSerializer serializer = new XmlSerializer(typeof(OrderEntity));

// Serialize an OrderEntity object to an XML file named "MyXmlOrderEntity.xml"
TextWriter writer = new StreamWriter("MyXmlOrderEntity.xml");
serializer.Serialize(writer, order);
writer.Close();
```

Serializing Objects in XML Web Services

The following code sample shows how to write an XML Web service that uses custom entity objects:

```
namespace MyWebService
{
  [WebService(Namespace="urn:MyWebServiceNamespace")]
  public class OrderWS : System.Web.Services.WebService
  {
    [WebMethod]
```

```
public OrderEntity GetOrder(int orderID)
{
 // Create an OrderDALC object.
 OrderDALC dalcOrder = new OrderDALC();

 // Use dalcOrder to get an OrderEntity object for the specified order ID.
 // This code assumes the OrderDALC class has a method named GetOrder,
 // which takes an Order ID as a parameter and returns an OrderEntity object
 // containing all the data for this order.
 OrderEntity order = dalcOrder.GetOrder(10248);

  // Return the OrderEntity object. The object will be serialized
  // automatically.
  return order;
}

[WebMethod]
public void UpdateOrder(OrderEntity order)
{
 // Create an OrderDALC object.
 OrderDALC dalcOrder = new OrderDALC();

 // Use dalcOrder to save the OrderEntity object's data to the database.
 // This code assumes the OrderDALC class has a method named UpdateOrder,
 // which receives an OrderEntity object and saves the data to the database.
 dalcOrder.UpdateOrder(order);
}
```

Note the following points concerning the preceding code:

- The GetOrder method receives an order ID as a parameter and returns an OrderEntity object that contains the data for this order.
- The UpdateOrder method receives an OrderEntity object and saves the object's data to the database.
- If the client application invokes the GetOrder and UpdateOrder methods, OrderEntity objects will automatically be serialized to XML format for the method call.

Default XML Format of a Serialized Custom Entity Object

The following XML document shows the default XML serialization format for OrderEntity objects:

```
<?xml version="1.0" encoding="utf-8"?>
<OrderEntity xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <OrderID>10248</OrderID>
  <CustomerID>VINET</CustomerID>
  <OrderDate>1996-07-04T00:00:00.0000000+01:00</OrderDate>
  <OrderDetails> ... see below ... </OrderDetails>
  <ShippedDate>1996-07-16T00:00:00.0000000+01:00</ShippedDate>
</OrderEntity>
```

The preceding document illustrates the default rules for XML serialization:

- The root element in the XML document is the same as the class name, OrderEntity.
- Each public property (and field) in the OrderEntity object is serialized to an element that has the same name.

The OrderDetails property in the OrderEntity class is a DataSet. DataSets provide built-in support for XML serialization. The OrderDetails DataSet is serialized as follows:

```
<OrderDetails>
  <xs:schema id="NewDataSet" xmlns=""
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true" msdata:Locale="en-UK">
      <xs:complexType>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="OrderDetails">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="OrderID" type="xs:int" minOccurs="0" />
                <xs:element name="ProductID" type="xs:int" minOccurs="0" />
                <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
                <xs:element name="Quantity" type="xs:short" minOccurs="0" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>
  <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
                   xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
    <NewDataSet>
      <OrderDetails diffgr:id="OrderDetails1" msdata:rowOrder="0"
                    diffgr:hasChanges="inserted">
        <OrderID>10248</OrderID>
        <ProductID>11</ProductID>
        <UnitPrice>14</UnitPrice>
        <Quantity>12</Quantity>
      </OrderDetails>
      <OrderDetails diffgr:id="OrderDetails2" msdata:rowOrder="1"
                    diffgr:hasChanges="inserted">
        <OrderID>10248</OrderID>
        <ProductID>42</ProductID>
        <UnitPrice>9.8</UnitPrice>
        <Quantity>10</Quantity>
      </OrderDetails>
      <OrderDetails diffgr:id="OrderDetails3" msdata:rowOrder="2"
                    diffgr:hasChanges="inserted">
        <OrderID>10248</OrderID>
```

```
        <ProductID>72</ProductID>
        <UnitPrice>34.8</UnitPrice>
        <Quantity>5</Quantity>
      </OrderDetails>
    </NewDataSet>
  </diffgr:diffgram>
</OrderDetails>
```

Note the following points concerning the serialization of DataSets:

- The <xs:schema> section describes the structure of the DataSet, including tables, column names, and column types.
- The <xs:diffgram> section contains the data of the DataSet. Each <OrderDetails> element represents a separate row in the OrderDetails table in the DataSet.

Controlling the XML Format of a Serialized Custom Entity Object

You can use .NET attributes in your custom entity class to control how the properties and fields are serialized to XML. Consider the following revised version of the OrderEntity class:

```
[XmlRoot(ElementName="Order", Namespace="urn:MyNamespace")]
public class OrderEntity
{
  [XmlAttribute(AttributeName="ID")]
  public int OrderID { ...get and set code, as before... }

  [XmlAttribute(AttributeName="CustID")]
  public string CustomerID { ...get and set code, as before... }

  [XmlElement(ElementName="Ordered")]
  public DateTime OrderDate { ...get and set code, as before... }

  public DataSet OrderDetails { ...get and set code, as before... }

  [XmlElement(ElementName="Shipped")
  public DateTime ShippedDate { ...get and set code, as before... }

  // Additional members, as required...
}
```

When an OrderEntity object is serialized to XML, it now has the following format:

```
<?xml version="1.0" encoding="utf-8" ?>
<Order ID="10248"
       CustID="VINET"
       xmlns="urn:MyNamespace"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Ordered>1996-07-04T00:00:00.0000000+01:00</Ordered>
```

```
  <OrderDetails> ...details same as before... </OrderDetails>
  <Shipped>1996-07-16T00:00:00.0000000+01:00</Shipped>
</Order>
```

For more information about how to use attributes to control XML serialization, see Attributes that Control XML Serialization.

How to Serialize Business Entity Components to SOAP Format

The following code sample shows how to use the SoapFormatter class to serialize an OrderEntity object to SOAP format. SOAP serialization also occurs (implicitly) when you pass an object to or from an XML Web service by using the SOAP protocol, and when you pass an object to or from a Remoting server by using an HTTP remoting channel. In addition, you can specify SOAP formatting when you use the TCP remoting channel.

```
using System.Runtime.Serialization.Formatters.Soap;    // For the SoapFormatter
class
...
// Create a SoapFormatter object, to serialize OrderEntity-type objects
SoapFormatter formatter = new SoapFormatter();

// Serialize an OrderEntity object to a SOAP (XML) file named
// "MySoapOrderEntity.xml"
FileStream stream = File.Create("MySoapOrderEntity.xml");
formatter.Serialize(stream, order);
stream.Close();
```

To use SOAP serialization for custom entity components, you must annotate your entity class by using the Serializable attribute, as shown in the following code:

```
[Serializable]
public class OrderEntity
{
  // Members, as before
```

If you want to customize the SOAP format generated during serialization, your entity class must implement the ISerializable interface. You must provide a GetObjectData method for the SoapFormatter to call during serialization, and you must provide a special constructor for the SoapFormatter to call to recreate the object during deserialization. The following code demonstrates the use of the ISerializable interface, the GetObjectData method, and the special constructor:

```
using System.Runtime.Serialization;   // For ISerializable interface, and related
types
...
[Serializable]
```

```
public class OrderEntity : ISerializable
{
  // Serialization function, called by the SoapFormatter during serialization
  void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctxt)
  {
    // Add each field to the SerializationInfo object
    info.AddValue("OrderID", orderID);
    // Additional code, as required...
  }

  // Deserialization constructor, called by the SoapFormatter during
deserialization
  public OrderEntity(SerializationInfo info, StreamingContext ctxt)
  {
    // Deserialize from the SerializationInfo object to the OrderEntity fields
    orderID = (int)info.GetValue("OrderID", typeof(int));
    // Additional code, as required...
  }

  // Other members, as before...
}
```

For more information about customized SOAP serialization, see Basic Serialization.

How to Serialize Business Entity Components to Binary Format

The following code sample shows how to use the BinaryFormatter class to serialize an OrderEntity object to binary format. Binary serialization also occurs (implicitly) when you pass an object to or from a Remoting server by using a TCP remoting channel. In addition, to improve performance, you can specify binary formatting when you use the HTTP remoting channel.

```
using System.Runtime.Serialization.Formatters.Binary;    // For the
BinaryFormatter class
...
// Create a BinaryFormatter object, to serialize OrderEntity-type objects
BinaryFormatter formatter = new BinaryFormatter();

// Serialize an OrderEntity object to a binary file named
// "MyBinaryOrderEntity.dat"
FileStream stream = File.Create("MyBinaryOrderEntity.dat");
formatter.Serialize(stream, order);
stream.Close();
```

To use binary serialization for custom entity objects, you must annotate your custom entity class by using the Serializable attribute. To customize the binary format generated during serialization, your custom entity class must implement the ISerializable interface. The coding details in both scenarios are the same as for SOAP serialization.

For more information about binary serialization, see Binary Serialization.