

Matisse .NET 2.0 Programmer's Guide

May 2009



Matisse .NET Programmer's Guide

Copyright ©1992–2009 Matisse Software Inc. All Rights Reserved.

Matisse Software Inc.
930 San Marcos Circle.
Mountain View, CA 94043
USA

Printed in USA.

This manual and the software described in it are copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without prior written consent of Matisse Software Inc. This manual and the software described in it are provided under the terms of a license between Matisse Software Inc. and the recipient, and their use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and international patents.

TRADEMARKS: Matisse and the Matisse logo are registered trademarks of Matisse Software Inc. All other trademarks belong to their respective owners.

PDF generated 28 May 2009

Content

1	Introduction	5
1.1	Scope of This Document	5
1.2	Reference Documentation	5
1.3	Before Running the Examples	5
1.4	Finding the Sample Code	5
2	Using ADO.NET	6
2.1	Running the Demo Program	6
2.2	Connecting to a Database Using ADO.NET	7
2.3	Executing an SQL Command	7
2.4	Retrieving Values or Objects using DataReader	8
2.5	Retrieving a Single Value	10
2.6	Calling Stored Methods	11
2.7	Filling a DataSet Using a DataAdapter	13
2.8	Establishing a Relationship between DataSet Tables	14
2.9	Transactions with ADO.NET	15
2.10	Matisse Data Types	15
3	Working with Objects and Values	17
3.1	Running ObjectsExample	17
3.2	Running ValuesExample	18
4	Working with Relationships	20
5	Working with Indexes	22
6	Working with Entry-Point Dictionaries	24
7	Connection and Transaction	25
7.1	Connection with Object Factory	25
7.2	Creating your Object Factory	26
7.3	Examples	26
8	Working with Versions	28
8.1	Building VerEx	28
8.2	Running VerEx	28
9	Building an Application from Scratch	31
9.1	Code Generation Using mt_dnom	31
9.2	Create a New Solution	31
9.3	Generate Schema Documentation	32
10	Code Generation and Data Classes	33
10.1	Code Generation	33
10.2	Data Classes	34

10.3	Copying Data	35
10.4	Storing Data Object's Updates into the Database	36
10.5	The mt_dnom Utility	37
11	Generating Class Stubs with a CodeDOM Provider	39
11.1	Building your Class Stub Generator	39
11.2	Example	39
Appendix A: Example.odl Schema		40
Appendix B: Managing a Database Schema with Object APIs		41
Appendix C: Browsing Database Objects with Visual Studio .NET		42
Index		43

1 Introduction

1.1 Scope of This Document

This document is intended to help .NET programmers learn the aspects of Matisse design and programming that are unique to the Matisse .NET binding.

Aspects of Matisse programming that the .NET binding shares with other interfaces, such as basic concepts and schema design, are covered in [Getting Started with Matisse](#). The document is available at <http://www.matisse.com/developers/documentation/>

Future releases of this document will add more advanced topics. If there is anything you would like to see added, or if you have any questions about or corrections to this document, please send e-mail to support@matisse.com.

Throughout this document, we presume that you already know the basics of .NET programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

1.2 Reference Documentation

The Matisse .NET binding installs the Reference Documentation in the Matisse docs directory of your installation (C:\Program Files\Matisse\docs\NET\MatisseNetBinding.chm by default). Refer to the reference documentation for the detailed information about constructors, properties, and methods of all the classes in the Matisse .NET binding.

1.3 Before Running the Examples

Before running the following examples, you must do the following:

- Install Matisse 8.x or higher.
- Install Visual Studio 2005 with .NET Framework 2.0 or higher

1.4 Finding the Sample Code

In this document, all the code examples are C# programs. However, both the C# and VB.NET sample codes discussed in this document are installed with the Matisse .Net binding, by default at C:\Program Files\Matisse\NET\Examples. The subdirectory names indicate which chapter(s) discusses the code.

2 Using ADO.NET

Matisse has a native data provider for ADO.NET, which is optimized for high performance and provides advanced features to dramatically simplify the persistent solution for your application.

The provider consists of classes which implement the standard interfaces for ADO.NET. These classes include:

- `com.matisse.Data.MtDatabase` - manages the connection to Matisse database
- `com.matisse.Data.MtCommand` - executes SQL statements or stored methods
- `com.matisse.Data.MtDataReader` - retrieves the data (values or objects) of executed command
- `com.matisse.Data.MtDataAdapter` - defines a set of data commands and database connection that are used to fill a DataSet and update the Matisse database.

A complete example program for ADO.NET is provided in the Matisse .NET binding installation. You can find it in `C:\Program Files\Matisse\NET\Examples` by default.

NOTE: Within Matisse, the ADO.NET provider is positioned as the standard way to execute SQL queries and stored methods, and to retrieve values or objects. Once you get these objects into your application from the database, without using Object-Relational mapping techniques, you will work with these objects using the object interface, which is provided as the Matisse .NET binding. The object interface provides high performance access to the database and better manageability of the persistent modeling. The object interface will be explained in the following chapters.

This chapter assumes that you are familiar with ADO.NET itself. If you are new to ADO.NET, Microsoft MSDN web site provides an introduction.

2.1 Running the Demo Program

Follow these instructions to run the ADO.NET demo program in the Matisse .NET installation directory, `C:\Program Files\Matisse\NET\Examples` by default.

1. Initialize the `example` database. Start the Matisse Enterprise Manager (Start --> Programs --> Matisse --> Enterprise Manager) and right click 'Start' on the `example` database. For more information, see the [Getting Started with Matisse](#) document.
2. Load the database schema into the database. From the Enterprise Manager, right click 'Schema->Load ODL Schema' on the `example` database. Then load the ODL (Object Definition Language) file `examples.odl` in the ADO example directory, by default it is located in `C:\Program Files\Matisse\NET\Examples\CSExamples\ADO`.
3. Open the `ADO.sln` file in Visual Studio .NET and build the solution.

4. Within a Command Prompt, run the built application with two arguments, your host name and the database name, which are `localhost` and `example` in this example.

The following sections will explain each feature of the demo program.

2.2 Connecting to a Database Using ADO.NET

The `MtDatabase` object provides the connectivity to a Matisse database. The following code shows two different ways to create a connection object, open the connection and close it:

```
[example 1]
MtDatabase dbcon = new MtDatabase("host1", "db1");
dbcon.Open();
// your code here ...
dbcon.Close();

[example 2]
string connectionString = "Server=host1;Database=db1";
MtDatabase dbcon = new MtDatabase(connectionString);
dbcon.Open();
// your code here ...
dbcon.Close();
```

The connection string can contain “User ID” and “Password” optionally.

2.3 Executing an SQL Command

After you open a connection to a Matisse database, you can execute commands (i.e., SQL statements or stored methods) and get the results from the database using an `MtCommand` object. You can create a command object for a specific `MtDatabase` object using the `CreateCommand` method of the `MtDatabase` object. You can also create a command object using the `MtCommand` constructors with various arguments.

You can use several methods to execute the specified SQL command for different purposes. The general guidelines for which method to use are:

- `ExecuteReader` - use this method when executing a `SELECT` statement that returns a table format result, or when executing a stored method using `Parameters` property.
- `ExecuteScalar` - use this method when executing a `SELECT` statement that returns a singleton value or when executing a stored method using the `CALL` syntax.
- `ExecuteNonQuery` - use this method when executing an `INSERT`, `UPDATE`, or `DELETE` statement or a DDL statement.

Example programs will be shown in the following sections.

2.4 Retrieving Values or Objects using DataReader

You use the `DataReader` object, which is returned by the `ExecuteReader` method, to retrieve values or objects from the database. Use the `Read` method to access each row in the result.

The `Read` method retrieves values or objects by chunk from the database server to increase the performance.

The following code demonstrates how to retrieve string and integer values from a `DataReader` object after executing a `SELECT` statement.

```
MtDatabase dbcon = new MtDatabase("your host", "your dbname");
// Open a connection to the database
dbcon.Open();

// Create an instance of MtCommand
IDbCommand dbcmd = dbcon.CreateCommand();

// Set the SELECT statement
dbcmd.CommandText = "SELECT LastName, Age FROM Person;";

// Execute the SELECT statement and get a DataReader
IDataReader reader = dbcmd.ExecuteReader();

string lname;
int age;
// Read rows one by one
while ( reader.Read() )
{
    // Get values for the first column
    lname = reader.GetString(0);

    // The second column 'Age' can be null. Check first if it is null or not.
    if ( ! reader.IsDBNull(1) )
        age = reader.GetInt32(1);
}

// Clean up and close the database connection
reader.Close();
dbcmd.Dispose();
dbcon.Close();
```

The Matisse data provider for ADO.NET provides a series of methods that allow you to retrieve column values in their native data types. These typed accessor methods are more efficient than the `GetValue` method. The next table lists the conversions between Matisse data types and .NET data types, as well as the accessor methods.

Table 2.4.1 Typed accessor methods of `DataReader`

Matisse data type	Accessor method of <code>DataReader</code>	Return type
MT_BYTE	<code>GetByte</code>	byte
MT_SHORT	<code>GetInt16</code>	short
MT_INTEGER	<code>GetInt32</code>	int
MT_LONG	<code>GetInt64</code>	long
MT_FLOAT	<code>GetFloat</code>	float
MT_DOUBLE	<code>GetDouble</code>	double
MT_NUMERIC	<code>GetDecimal</code>	decimal
MT_STRING (ASCII)	<code>GetString</code>	string
MT_STRING (Unicode)	<code>GetUnicode</code>	string
MT_CHAR	<code>GetChar</code>	char
MT_BOOLEAN	<code>GetBoolean</code>	bool
MT_DATE	<code>GetDateTime</code> , <code>GetDate</code>	DateTime
MT_TIMESTAMP	<code>GetDateTime</code> , <code>GetTimestamp</code>	DateTime
MT_INTERVAL	<code>GetInterval</code>	TimeSpan
MT_OID	<code>GetObject</code>	MtObject
MT_NULL		DBNull.Value
MT_ANY	<code>GetValue *</code>	Object

* It is recommended to use a specific typed accessor method if you know the data type in advance.

Retrieving Objects

You can retrieve C# objects (or VB objects) directly from the database without using the Object-Relational mapping technique. This method eliminates the unnecessary complexity in your application, i.e., O/R mapping layer, and improves your application performance and maintenance.

To retrieve objects, use `REF` in the select-list of the query statement and the `GetObject` method returns an object. The following code example shows how to retrieve `Person` objects using a `DataReader` object.

```
// Create an instance of MtCommand using the connection object dbcon
IDbCommand dbcmd = dbcon.CreateCommand();

// Set the SELECT statement
dbcmd.CommandText = "SELECT REF(p) FROM Person p WHERE LastName =
'Watson'";

// Execute the SELECT statement and get a DataReader
```

```

MtDataReader reader = (MtDataReader) dbcmd.ExecuteReader();

Person obj;
// Read rows one by one
while ( reader.Read() )
{
    // Get the C# object from the current row
    obj = (Person) reader.GetObject(0);
}

// Clean up and close the database connection
reader.Close();
dbcmd.Dispose();

```

For more information about the persistent classes and their code generation, e.g., Person class in the above example, see the chapters [3 Working with Objects and Values](#) and [9 Building an Application from Scratch](#).

CAUTION: If your persistent classes are defined in a specific namespace, i.e., not in the anonymous default namespace, or in a separate assembly, you need to pass this information to the `Connection` object. Otherwise, you will get an `InvalidCastException` when you are retrieving objects, e.g., when calling the `GetObject` method. See the [7.1 Connection with Object Factory](#).

Using Parameters

You can use the `Parameters` collection to explicitly define parameters for an SQL statement. In the following example, the where-clause condition uses a named parameter `@lname` for Person's `LastName`.

```

// Set the SELECT statement
dbcmd.CommandText =
    "SELECT FirstName, LastName FROM Person p WHERE LastName = @lname;";

// Set the parameter for @lname with type information
dbcmd.Parameters.Add("@lname", MtType.MtBasicType.STRING).Value = "Watson";

// Execute the SELECT statement and get a DataReader
IDataReader reader = dbcmd.ExecuteReader();

```

All the Matisse data types are listed in [2.10 Matisse Data Types](#).

2.5 Retrieving a Single Value

When you need to obtain a single value from a database, e.g., executing an aggregate function `COUNT(*)` with an `SELECT` statement, you can use the `ExecuteScalar` method of the `Command` object. The `ExecuteScalar` method returns a value of the first column of the first row when executing a `SELECT` statement.

The following example shows how to get the total number of instances of class `Person` in the database.

```

// Create a command object from a connection object
MtCommand dbcmd = dbcon.CreateCommand();

// Set the query statement
dbcmd.CommandText = "SELECT COUNT(*) FROM Person;";

// Execute the query, and get aggregate value
int cnt = (int) dbcmd.ExecuteScalar();

// clean up
dbcmd.Dispose();

```

2.6 Calling Stored Methods

You can call a stored method using the `CALL` syntax, i.e., simply passing the stored method name followed by arguments as an SQL statement. ADO.NET provides an advanced feature to call a stored method, which allows you to explicitly specify the method's parameters and return values using the `Parameters` collection and `Command` object.

Calling Stored Methods Using Parameters Collection

To call a stored method using `Parameters` collection, set the `CommandType` of the `Command` object to `StoredProcedure`. Then, you can use `Parameters` collection to set parameters and a return value for a stored method.

For the `CommandType` `StoredProcedure`, we allow only positional parameters, not named parameters. A return value needs to be added first, then method arguments follow.

The following program code shows how to call the stored method `CountByLName` of the `Person` class, which is generated by the sample ADO.NET program in the Matisse .NET installation.

```

// Create a command object from a connection object
MtCommand dbcmd = dbcon.CreateCommand();

// Specify the stored method. Since it is a static method that we will call,
// the name is consisted of class name and method name.
dbcmd.CommandText = "Person::CountByLName";
dbcmd.CommandType = CommandType.StoredProcedure;

// Set the parameter for Return Value
MtParameter retParam =
    dbcmd.Parameters.Add("@ObjCount", MtType.MtBasicType.INTEGER);
retParam.Direction = ParameterDirection.ReturnValue;

// Set the first parameter for the method
dbcmd.Parameters.Add("@lastname", MtType.MtBasicType.STRING);
dbcmd.Parameters["@lastname"].Value = "Watson";

//Execute the stored method
dbcmd.ExecuteNonQuery();

```

```
// Get the returned value
int count = (int) dbcmd.Parameters["@ObjCount"].Value;

// clean up and close the connection
dbcmd.Dispose();
```

Calling Stored Methods Using CALL syntax

You can call a stored method using the `CALL` syntax, without using `Parameters` collection. Use the `ExecuteScalar` method of the `Command` object to execute the `CALL` statement. The method can return a value, such as an integer, a string, or a list of timestamp, an object or a list of objects.

The next sample code illustrates how to call the stored method `FindByName` of the `Person` class, which is generated by the sample ADO.NET program in the Matisse .NET installation. Note that the method returns a `Person` object, and you receive the object immediately without any mapping technique, e.g., O/R mapping. Also note that the method name contains both the class name and the method name, since it is a static method.

```
// Create a command object from a connection object
MtCommand dbcmd = dbcon.CreateCommand();

// Use CALL syntax to call the method
dbcmd.CommandText = "CALL Person::FindByName('Watson', 'James');";

// Execute the stored method, and get the returned object
Person p = (Person) dbcmd.ExecuteScalar();

// Clean up
dbcmd.Dispose();
```

Returning a List of Objects from a Stored Method

You can return a list of objects, called a `Selection`, from a stored method using the `CALL` syntax. You use the same method just described above, but need a “correct” casting of the result.

```
dbcmd.CommandText = "CALL Person::FindPersonsByLastName('Watson');";

// Execute the stored method, and get a list of objects
object[] persons = (object[]) dbcmd.ExecuteScalar();
Person aPerson = (Person) persons[0];
...
```

The SQL method can be defined as following, for example:

```
CREATE STATIC METHOD FindPersonsByLastName (name STRING)
RETURNS SELECTION(Person)
FOR Person
BEGIN
    DECLARE res SELECTION(Person);
    SELECT REF(p) FROM Person p WHERE LastName = name INTO res;
    RETURN res;
END;
```

2.7 Filling a DataSet Using a DataAdapter

A `DataAdapter` object, an instance of `MtDataAdapter` in the Matisse ADO.NET provider, retrieves data from a database and populates tables with rows within a `DataSet` object. The `SelectCommand` property of the `DataAdapter` object needs to be set to retrieve data from the database. The `DataAdapter` object uses the `Connection` object, which is usually passed as an argument when the `DataAdapter` object is constructed, to connect to the database. If the database connection is not open, the `DataAdapter` object opens the connection, retrieves data, and then closes the connection. If the database connection is already open, the connection remains open after the `DataAdapter` object retrieves data.

Use the `Fill` method of the `DataAdapter` to populate a `DataSet` with the result of a `Command` execution. The following code sample demonstrates how to fill a table “Persons” with all the rows returned by the `SELECT` statement.

```
// Create a new DataAdapter to get all the Person objects
MtDataAdapter myDA = new MtDataAdapter ("SELECT * FROM Person", dbcon);

// Create a new DataSet
DataSet myDS = new DataSet();

// Fill the table "Persons" with rows selected by the SELECT statement
myDA.Fill(myDS, "Persons");

// Get the table "Persons"
DataTable personTable = myDS.Tables["Persons"];

// Get each row in the table
foreach (DataRow row in personTable.Rows)
{
    // Get the values of each column, and print them
    Console.WriteLine(((string) row["OID"]).PadRight(10) +
        ((string) row["FirstName"]).PadRight(20) +
        ((string) row["LastName"]).PadRight(20));

    // The column Age can be null. We need to check it.
    if ( row.IsNull("Age") )
        Console.WriteLine("NULL");
    else
        Console.WriteLine((int) row["Age"]);
}
```

NOTE: The `DataSet` object caches data locally in your application and performs processing on the data without an open connection to the database. It does improve the concurrency of the Matisse database server. In addition to this, Matisse provides a version access (read-only transaction) mechanism to access a Matisse database without locking objects. Using version access (and without the `DataSet`), you can directly work on objects, not the table format data only, which eventually

simplifies your application and gives better performance. For more information about version access, see the *Getting Started with Matisse* document.

2.8 Establishing a Relationship between DataSet Tables

You can relate one table to another in a `DataSet` object, in order to navigate through tables, using a `DataRelation` object.

The following code example adds a relationship between the `Manager` table and the `Employee` table, and lists all the employees for each manager.

```

DataSet testDS = new DataSet();

// Parent; fill the managers table
MtDataAdapter adapter1 = new MtDataAdapter ("SELECT OID, FirstName, LastName
FROM Manager;", dbcon);
adapter1.Fill (testDS, "managers");

// Children; fill the employees table
MtDataAdapter adapter2 =
    new MtDataAdapter ("SELECT OID, FirstName, LastName, ReportsTo.OID
boss_OID FROM Employee;", dbcon);
adapter2.Fill (testDS, "employees");

// Create a relationship between the two tables
testDS.Relations.Add ("Team",
    testDS.Tables["managers"].Columns["OID"],
    testDS.Tables["employees"].Columns["boss_OID"]);

// Read the tables using the parent-child relationship
string mgrFName, mgrLName, fname, lname;
foreach (DataRow parentRow in testDS.Tables["managers"].Rows)
{
    mgrFName = (string) parentRow["FirstName"];
    mgrLName = (string) parentRow["LastName"];

    // Read all the employees for each manager
    foreach (DataRow childRow in
parentRow.GetChildRows (testDS.Relations["Team"])) {
        fname = (string) childRow["FirstName"];
        lname = (string) childRow["LastName"];

        Console.WriteLine(mgrFName + " " + mgrLName + ", " + fname + " " +
lname);
    }
}

```

NOTE: Matisse natively supports relationships between objects, which can simplify your application and perform much faster. For more information, see [4 Working with Relationships](#).

2.9 Transactions with ADO.NET

In the example programs described in the preceding sections, transactions are not explicitly started, but are started implicitly by SQL statements. These transactions are committed implicitly when the connection is closed, unless you terminate the transactions explicitly.

NOTE: Precisely, when you execute an SQL statement with no updates to the database, i.e., `SELECT` statement, a read-only transaction (version access) is started if no transaction is started. When you execute an `UPDATE`, `INSERT`, `DELETE`, or a DDL statement, a transaction is started.

You can start, commit, or rollback a transaction and read-only transaction (version access) explicitly using a `Connection` object and a `Transaction` object. We recommend you to manage transactions explicitly, since it gives you better control over transactions span.

The following code shows transactions using ADO.NET.

```
// Open a connection to the database
MtDatabase dbcon = new MtDatabase("your host", "your dbname");
dbcon.Open();

// Start a transaction
IDbTransaction dbtran = dbcon.BeginTransaction();

// Create an instance of MtCommand and set an INSERT statement
IDbCommand dbcmd = dbcon.CreateCommand();
dbcmd.CommandText =
    "INSERT INTO Person (FirstName, LastName) VALUES ('John', 'Doe');";

// Execute the SQL statement
dbcmd.ExecuteNonQuery();

// Commit the transaction
dbtran.Commit();
```

2.10 Matisse Data Types

The table below lists all the Matisse data types and their corresponding `enum` values in the Matisse .NET binding. These database type values are used with `Parameter` object, for example:

```
// Set the parameter for @lname with type information
dbcmd.Parameters.Add("@lname", MtType.MtBasicType.STRING).Value = "Watson";
```

Table 2.10.1 Matisse data types

Matisse Data Type	.NET database type
MT_ANY	MtType.MtBasicType.ANY
MT_AUDIO	MtType.MtBasicType.AUDIO
MT_BOOLEAN	MtType.MtBasicType.BOOLEAN
MT_BOOLEAN_LIST	MtType.MtBasicType.BOOLEAN_LIST
MT_BYTE	MtType.MtBasicType.BYTE
MT_BYTES	MtType.MtBasicType.BYTES
MT_CHAR	MtType.MtBasicType.CHAR
MT_DATE	MtType.MtBasicType.DATE
MT_DATE_LIST	MtType.MtBasicType.DATE_LIST
MT_DOUBLE	MtType.MtBasicType.DOUBLE
MT_DOUBLE_LIST	MtType.MtBasicType.DOUBLE_LIST
MT_FLOAT	MtType.MtBasicType.FLOAT
MT_FLOAT_LIST	MtType.MtBasicType.FLOAT_LIST
MT_INTEGER	MtType.MtBasicType.INTEGER
MT_INTEGER_LIST	MtType.MtBasicType.INTEGER_LIST
MT_INTERVAL	MtType.MtBasicType.INTERVAL
MT_INTERVAL_LIST	MtType.MtBasicType.INTERVAL_LIST
MT_LONG	MtType.MtBasicType.LONG
MT_LONG_LIST	MtType.MtBasicType.LONG_LIST
MT_NULL	MtType.MtBasicType.NULL
MT_NUMERIC	MtType.MtBasicType.NUMERIC
MT_NUMERIC_LIST	MtType.MtBasicType.NUMERIC_LIST
MT_OID	MtType.MtBasicType.OID
MT_SELECTION	MtType.MtBasicType.SELECTION
MT_SHORT	MtType.MtBasicType.SHORT
MT_SHORT_LIST	MtType.MtBasicType.SHORT_LIST
MT_STRING	MtType.MtBasicType.STRING
MT_STRING_LIST	MtType.MtBasicType.STRING_LIST
MT_TEXT	MtType.MtBasicType.TEXT
MT_TIMESTAMP	MtType.MtBasicType.TIMESTAMP
MT_TIMESTAMP_LIST	MtType.MtBasicType.TIMESTAMP_LIST
MT_VIDEO	MtType.MtBasicType.VIDEO

3 Working with Objects and Values

This chapter and the following chapters will explain the object interface of the Matisse .NET binding. The object interface allows you to directly retrieve objects from the Matisse database without Object-Relational mapping, navigate from one object to another through the relationship defined between them, and update properties of objects without writing SQL statements.

The object interface can be used with ADO.NET. For example, you can retrieve objects using ADO.NET **Command** object, as explained in [2.4 Retrieving Values or Objects using DataReader](#), then use the object interface to navigate to other objects from these objects, or update properties of these objects using the accessor methods defined on these classes.

This chapter and the following chapters use example programs installed with the Matisse .NET binding, at `C:\Program Files\Matisse\NET\Examples` by default. These examples already contain persistent classes, which are generated by the `mt_dnom` utility. For more information about the code generation of persistent classes, refer to [9 Building an Application from Scratch](#).

3.1 Running ObjectsExample

`ObjectsExample` shows the following features:

- Creation of new persistent objects
- Retrieve all objects of a class, including all its subclasses
- Delete persistent objects

Follow the next instructions to run the example program:

1. Create and initialize a database named `example`, or whatever you like, as described in [Getting Started with Matisse](#).
2. In a command-line window, change to the appropriate `chap_3` directory and load into the database the database schema `objects.odl`, which is an ODL (Object Definition Language) file:

```
> mt_sdl -d example import -odl objects.odl
```

3. Open `Chap_3.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `ObjectsExample` and `ValuesExample` applications.
4. For C# only, you may generate HTML documentation for the schema using NDoc, a free utility downloadable from ndoc.sourceforge.net. Select `Schema\bin\Debug\Schema.dll` as the Assembly Filename and NDoc will automatically set the `Schema.xml` file in the same directory as the XML Doc Filename.
5. In a command-line window, change to the `ObjectsExample\bin\Debug` (for C#) or `ObjectsExample\bin` (for VB) directory and run the application:

```
> ObjectsExample localhost example
```

See the comments in `ObjectsExample.cs` or `ObjectsExample.vb` for additional information. The following code is an excerpt from these files with some more comments.

```
// Open a database connection and start a transaction.
// Note: In this example, the persistent classes are defined in a separate
//       assembly 'Schema'.
//       We pass a MtPackageObjectFactory object to the connection object so
//       that the connection object can find the persistent classes in the
//       assembly named 'Schema'.
//       For more information, look at 7.1 Connection with Object Factory.
MtDatabase db = new MtDatabase("host", "db", new
MtPackageObjectFactory(", Schema"));
db.Open();
db.BeginTransaction();

// Create a new persistent object of class Person, and set its property using
the
// accessor method.
// Note that you need to set all the non-nullable properties of the object
before
// you commit the transaction.
Person p = new Person(db);
p.FirstName = "John";

// List all the object of class Person and its all subclasses, i.e., Employee
// and Manager.
foreach (Object obj in Person.InstanceEnumerator(db))
{
    Person psn = (Person) obj;
}

// Remove the persistent object from the database
p.Remove();

// Commit the transaction and close the database connection
db.Commit();
db.Close();
```

3.2 Running ValuesExample

This example shows how to get and set values for various Matisse data types including Null values, and how to check if a property of an object is a Null value or not.

ValuesExample uses the database created for ObjectsExample. It creates an object, then manipulates its values in various ways as described in the source-code comments.

To run the application, open a command-line window, change to the `ValuesExample\bin\Debug` (for C#) or `Example\bin` (for VB) directory, and enter:

```
> ValuesExample localhost example
```

Here are some excerpts from the source code.

```
// Open a database connection and start a transaction so we can create new
// persistent objects.
// For more information on how to use MtPackageObjectFactory, look at 7.1
Connection with Object Factory
MtDatabase db = new MtDatabase("host", "db", new
MtPackageObjectFactory(", Schema"));
db.Open();
db.StartTransaction();

// Create a new persistent object of class Employee
Employee e = new Employee(db);

// Setting strings
e.Comment = "setting values";

// Setting an integer
e.Age = 42;

// Setting a date
e.HireDate = new DateTime(2002, 5, 31);

// Setting a decimal number
e.Salary = new Decimal(3958.33);

// Check if the Employee object's Age property is NULL or not
if (!e.IsNull(Employee.GetAgeAttribute(db)))
    Console.WriteLine("\t" + e.Age + " years old");

// Two different ways to remove a property value, i.e., set NULL
e.Comment = null;
e.SetNull(Employee.GetAgeAttribute(db));
```

See the comments in `ValuesExample.cs` or `ValuesExample.vb` for more information.

4 Working with Relationships

One of the big advantages of the object interface of the Matisse .NET binding is the ability to navigate from one object to another through a relationship defined between them. It is as easy as you access a property of an object. You don't have to use complicated SQL joins nor many `DataRelation` objects in a `DataSet` in ADO.NET. For more information about Matisse relationships, see the section [Referential Integrity and Cardinality Constraints](#) in [Getting Started with Matisse](#).

`RelationshipsExample` creates several objects, then manipulates the relationships among them in various ways as described in the source-code comments. To run the sample program, follow the next instructions.

1. Create and initialize a database named `example` as described in [Getting Started with Matisse](#).
2. In a command-line window, change to the appropriate `chaps_4_5_6` directory and load into the database the database schema `examples.odl`, which is an ODL (Object Definition Language) file.

```
> mt_sdl -d example import -odl examples.odl
```

3. Open `Chaps_4_5_6.sln` in Visual Studio .NET and select Build / Build Solution. This will generate the schema class files defined in `examples.odl` and compile the `RelationshipsExample`, `IndexExample`, and `EPDictExample` applications.
4. For C# only, you may generate HTML documentation for the schema using NDoc, a free utility downloadable from ndoc.sourceforge.net. Select `Schema\bin\Debug\Schema.dll` as the Assembly Filename and NDoc will automatically set the `Schema.xml` file in the same directory as the XML Doc Filename.
5. In a command-line window, change to the `RelationshipsExample\bin\Debug` (for C#) or `RelationshipsExample\bin` (for VB) directory and run the application:

```
> RelationshipsExample localhost example
```

The following code illustrates the key points of the example program.

```
// Create persistent objects
Manager m1 = new Manager(db);
Employee e1 = new Employee(db);

// Set a relationship.
// You can use a simple assignment expression, since the maximum cardinality
// of the 'Assistant' relationship is 1.
m1.Assistant = e1;

// Create other objects and set relationship 'Assistant' between them
Manager m2 = new Manager(db);
Employee e2 = new Employee(db);
m1.Assistant = e2;
m2.Assistant = e2;
```

```
// The relationship 'assistantOf' is the inverse relationship of
'Assistant'.
// When a relationship is updated, its inverse relationship is automatically
updated
// to enforce referential integrity.
// Even though we didn't update 'AssistantOf' explicitly, the next lines
// return all the managers that e2 is assisting.
foreach (Manager manager in e2.AssistantOf)
    Console.WriteLine(manager.FirstName + manager.LastName);

// Create two Person objects, which will be added to the 'Children'
relationship of m2
Person c1 = new Person(db);
Person c2 = new Person(db);

// Set the relationship 'Children' with two Person objects c1 and c2
m2.Children = (new Person[] {c1, c2});

// Create another Person object and append it to the end of the 'Children'
relationship
Person c3 = new Person(db);
m2.AppendChildren(c3);

// Iterating through all the children of m2
foreach (Object o in m2.ChildrenEnumerator())
    Console.WriteLine(((Person)o).FirstName);
```

See the comments in RelationshipsExample.cs or RelationshipsExample.vb for additional information.

5 Working with Indexes

While indexes are used mostly by the SQL query optimizer to speed up queries, the Matisse .NET binding also provides the index query APIs to look up objects based on a key value(s). The lookup APIs are defined in persistent classes as static methods by the `mt_dnom` code generator.

For example, the class `Person` has an index `personName` defined on the two attributes `firstName` and `lastName`, and this will generate several methods in the `Person` class file for C#:

```
static Person LookupPersonName (MtDatabase db,
                                string lastName,
                                string firstName)

static MtObjectEnumerator PersonNameEnumerator (MtDatabase db,
                                                string fromLastName,
                                                string fromFirstName,
                                                string toLastName,
                                                string toFirstName)

static MtObjectEnumerator PersonNameEnumerator (MtDatabase db,
                                                string fromLastName,
                                                string fromFirstName,
                                                string toLastName,
                                                string toFirstName,
                                                MtClass filterClass,
                                                MtIndex.MtDirection direction,
                                                int numObjPerBuffer)
```

The first method returns a `Person` object when an object matches both the first name and the last name. The method returns null if no object matches, or raises a Matisse exception if more than one object match the criteria.

The second and the third methods do a range query based on the starting value and ending value, and return an `Enumerator` object, with which you can iterate through all the matched objects.

The third one is identical with the second one except that it offers more options to tweak the query condition:

- `filterClass` is used to filter the result by class hierarchy. For example, if you set class `Employee`, you will get only objects from `Employee` class or `Manager` class, excluding proper `Person` objects.
- `direction` is either 'as it is' or 'reverse', specified by `MtIndex.MtDirection.DIRECT` or `MtIndex.MtDirection.REVERSE`. `MtIndex.MtDirection.DIRECT` is the default value.
- `NumObjPerBuffer` is the number of matched objects transferred from the server to the client by each call.

Note that the same sort of methods are generated for other languages as well.

Range Query with Index

With the starting value(s) and the ending value(s), you can execute a range query as shown above. The starting value(s) needs to be smaller than (or equal to) the ending value(s).

You can also execute a range query with an open range using the index query APIs, for example, selecting `Person` objects who are older than or equal to 21. Suppose you have defined an index `AgeIdx` for the `age` attribute, then you would write a code for the query:

```
anEnumerator = Employee.AgeIdxEnumerator(db, 21, Int32.MaxValue);
```

For string, you can use `null` to specify an open end.

Running IndexExample

The sample program `IndexExample` uses the database created for `RelationshipsExample`. Using the `PersonName` index, it checks whether the database contains an entry for a person matching the specified first name and the last name.

To run the application, open a command-line window, change to the `IndexExample\bin\Debug` (for C#) or `IndexExample\bin` (for VB) directory, and enter:

```
> IndexExample localhost example firstName lastName
```

The application will list the names of `Person` objects in the database, indicate whether the name specified as arguments to the command was found. And also it will return the result of a range query using an enumerator.

See the comments in `IndexExample.cs` or `IndexExample.vb` for additional information.

6 Working with Entry-Point Dictionaries

An entry-point dictionary is an indexing structure containing keywords derived from a value, which is especially useful for full-text indexing. While the entry-point dictionary can be used with SQL query using `ENTRY_POINT` keyword, the object interface of the Matisse .NET binding also provides APIs to directly retrieve objects using the entry-point dictionaries.

For example, the `Person` class has an full-text entry-point dictionary `commentDict` defined on the attribute `comment`, and this will generate several methods in the `Person` class file.

```
static Person LookupCommentDict (MtDatabase db, string ep)
static MtObjectEnumerator CommentDictEnumerator (MtDatabase db, string ep)
static MtObjectEnumerator CommentDictEnumerator (MtDatabase db, string ep,
                                                MtClass filterClass,
                                                int numObjPerBuffer)
```

The first method returns a `Person` object when one object, and only one object, matches the criteria. The method returns null if no object matches, or raises a Matisse exception if more than one object match the criteria.

The second and the third method return an Enumerator object, which allows you to iterate through all the `Person` objects that match the criteria.

The third method is identical with the second one except that it offers more options to tweak the query condition. `FilterClass` is used to filter the result by class hierarchy. For example, if you set the class `Employee`, you will get only objects from `Employee` class or `Manager` class, excluding proper `Person` objects. `NumObjPerBuffer` is the number of matched objects transferred from the server to the client by each call.

For more information about the entry-point dictionary, see the section [Entry-Point Dictionaries](#) in [Getting Started with Matisse](#) document.

Running EPDictExample

`EPDictExample` uses the database created for `RelationshipsExample`. Using the `commentDict` entry-point dictionary, it counts the number of `Person` objects in the database with `Comments` fields containing the specified string.

To run the application, open a command-line window, change to the `EPDictExample\bin\Debug` (for C#) or `EPDictExample\bin` (for VB) directory, and enter:

```
> EPDictExample localhost example search_string
```

See the comments in `EPDictExample.cs` or `EPDictExample.vb` for additional information.

7 Connection and Transaction

All interactions between client .NET applications and Matisse databases take place within the context of transactions (either explicit or implicit) established by database connections, which are transient instances of the `MtDatabase` class. Once the connection is established, your .NET application may interact with the database using the schema-specific methods generated by `mt_dnom` or ADO.NET.

7.1 Connection with Object Factory

Using `MtPackageObjectFactory`

When your persistent classes are defined in a specific name space or in a separate assembly, you need to give these information to the `Connection` object so that the `Connection` object can find these classes when returning objects.

For example, in the Chapter 3 example, the persistent classes are defined in the separate assembly `Schema` without any specific namespace. In this case, you need to pass an `MtPackageObjectFactory` object as the additional argument for the `MtDatabase` constructor.

```
MtDatabase db = new MtDatabase("host", "db", new
MtPackageObjectFactory(", Schema"));
```

The string format for the argument of `MtPackageObjectFactory` constructor is as follows.

<namespace>, <assembly name>

`<namespace>` can be omitted when the persistent classes are defined without specific namespace, i.e., in the anonymous default namespace. Note that you cannot omit “,” (comma) when you omit `<namespace>`. For example, if the persistent classes are in the namespace `Matisse.Project1` in the assembly `proj1.dll`, then the constructor would look like

```
new MtPackageObjectFactory("Matisse.Project1,proj1")
```

If the persistent classes are in the multiple namespaces or multiple assemblies, you pass a list of strings to the `MtPackageObjectFactory` constructor,

```
new MtPackageObjectFactory(new string[]
{"Matisse.Project1,proj1", "Matisse.Project2,proj2"})
```

By default, the anonymous default namespace in the startup assembly is searched as well as the specified namespaces.

Using `MtCoreObjectFactory`

This factory is the basic `MtObject`-based object factory. This factory is the most appropriate for application which does use generated stubs. This factory is faster than the default Object Factory used by `MtDatabase` since it doesn't use reflection to build objects.

```
MtDatabase db = new MtDatabase("host", "db", new MtCoreObjectFactory());
```

7.2 Creating your Object Factory

Implementing the MtObjectFactory interface

The `MtObjectFactory` interface describes the mechanism used by `MtDatabase` to create the appropriate .NET object for each Matisse object. Implementing the `MtObjectFactory` interface requires to define the `GetObjectInstance()` method which return a .NET object based on an oid.

```
class MyAppFactory : MtObjectFactory
{
    public object GetObjectInstance(MtDatabase database, int oid)
    {
        return new MtObject(database, oid);
    }
}
```

Implementing a Sub-Class of MtCoreObjectFactory

This `MtCoreObjectFactory` is a basic `MtObject`-based object factory which can be extended to implement your own Object Factory.

```
class MyAppFactory : MtCoreObjectFactory
{
    public override object GetObjectInstance(MtDatabase database, int oid)
    {
        if (IsSchemaObject(database, oid))
        {
            return base.GetObjectInstance(database, oid);
        }
        else
        {
            // Create your .NET object as you see fit
            return anObject;
        }
    }
}
```

7.3 Examples

This section provides four example programs. The following table lists these examples with their description.

Project Name	Description
Connect	This example demonstrates the basic features such as establishing a connection to a database, starting and committing a transaction, and closing the connection.
VersionConnect	This example shows how to start a read-only transaction (version access), which is suitable for online analysis or reporting purposes.
VersionNavigation	This one illustrates the method to access saved versions in a database. For more information about saved version, see the section Matisse in Operation in the Getting Started with Matisse document.
AdvancedConnect	This example shows how to enable the memory transport in the local client-server connection to speed up the communication. It also demonstrates how to get and set the diverse connection options such as data access mode or transport type.

To run the programs,

- Create and initialize a database
- Build the solution in Visual Studio .NET
- Within a “Command Prompt” window, Go to each bin\Debug directory in each project, and run the command with two arguments <hostname> and <database name>.

And also, see the comments in the source files (e.g., VersionConnect.cs or VersionConnect.vb) for more information.

8 Working with Versions

This section is for advanced users. If you are a first-time user, you may skip to the next section.

This example is a very simple demonstration of version control, i.e., save the versions for later access. For more information about version access and named versions (savetimes), refer to [Getting Started with Matisse](#). The example uses a simple one-class schema with three attributes (represented here in ODL, Object Definition Language):

```
attribute Integer id;
mt_entry_point_dictionary id entry_point_of id;
attribute String FirstName = "(unset)";
attribute String LastName = "(unset)";
```

VerEx allows you to create and modify objects. Whenever an object is created or modified, the application automatically creates a new named version (savetime). To create an object, pass VerEx the following command, specifying a new `id` value:

```
set id attribute_name value
```

To modify an object, enter the same command, specifying the `id` value of the object. To list all versions, objects, and values, enter the command `dump`.

See the comments in `VerEx.cs` or `VerEx.vb` for additional information.

8.1 Building VerEx

1. Follow the instructions in [Before Running the Examples](#) on page 5.
2. Create and initialize a database as described in [Getting Started with Matisse](#).
3. In a command-line window, change to the appropriate `Versions` directory and load `VerEx.odl` into the database:

```
> mt_sdl -d example import -odl VerEx.odl
```

4. Open `Versions.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the VerEx application.

8.2 Running VerEx

1. Open a Command Prompt window, change to the `VerEx\bin\Debug` (for C#) or `VerEx\bin` (for VB) directory, and enter:

```
> VerEx example set 1 FirstName John
```

This creates an object with `id=1`, `FirstName=John`, and `LastName unset`, and command-line output similar to the following (your version name may vary):

```
New Version VerEx00000005 created
```

You can verify this by entering the following command:

```
> VerEx example dump
```

Which will produce output similar to the following (your version name may vary):

```
Version VEREX00000005
  1: John (unset)
```

2. Enter the following command to modify the object:

```
> VerEx example set 1 LastName Smith
```

This creates a second version of the object, with `LastName= Smith`. You can see this by repeating the dump command shown in step 1:

```
Version VerEx00000005
  1: John (unset)
Version VEREX100000007
  1: John Smith
```

3. Modify the object again:

```
> VerEx example set 1 FirstName Jack
```

This creates a third version of the object, with `FirstName= Jack`. You can see this by repeating the dump command:

```
Version VEREX00000005
  1: John (unset)
Version VEREX00000007
  1: John Smith
Version VEREX00000009
  1: Jack Smith
```

4. Now add a second object:

```
> VerEx example set 2 FirstName Jane
```

5. And modify the second object:

```
> VerEx example set 2 LastName Jones
```

6. Run the dump command again and you will see that there are now four versions of the database, one for each modification you made:

```
Version VEREX00000005
  1: John (unset)
Version VEREX00000007
  1: John Smith
Version VEREX00000009
  1: Jack Smith
Version VEREX0000000B
  2: Jane (unset)
  1: Jack Smith
Version VEREX0000000D
```

2: Jane Jones

1: Jack Smith

You can also view a list of these versions in the Enterprise Manager, by selecting “Database Snapshots” under the Data node.

9 Building an Application from Scratch

9.1 Code Generation Using mt_dnom

1. Design a database schema using ODL (Object Definition Language), SQL DDL, or IBM-Rational Rose UML designer, and load it into a new database as described in [Getting Started with Matisse](#). For more information about each method, see [Matisse ODL Programmer's Guide](#) for ODL, [Matisse SQL Programmer's Guide](#) for SQL DDL, or [Matisse Rose Link User's Guide](#) for Rational Rose.
2. Open a command-line window and change to the .NET project directory.
3. Enter the following command to generate C# code:

```
> mt_dnom -d mydbname stubgen -lang C#
```

or the following command to generate VB.NET code:

```
> mt_dnom -d mydbname stubgen -lang VB
```

A `.cs` or `.vb` file will be created for each class defined in the database. If you need to define these persistent classes in a specific namespace, use `-n` option. The following command generates classes under the namespace `MyCompany.MyProject`:

```
> mt_dnom -d mydbname stubgen -lang C# -n MyCompany.MyProject
```

When you update your database schema later, load the updated schema into the database using your favorite method (`mt_sdl`, SQL, or Rational Rose). Then, execute the `mt_dnom` utility in the directory where you first generated the class files, to update the files. Your own program codes added to these class files will be preserved.

NOTE: The classes generated by the above command are sometimes called “stub classes” to make them distinguished from “data classes”, which are explained in [10 Code Generation and Data Classes](#).

9.2 Create a New Solution

1. Open Visual Studio .NET and create a new Visual Studio solution.
2. Set the project to reference `MatisseNet.dll`, the Matisse library: right-click on the project icon, select Add Reference, select the Projects tab, click Browse, navigate to `c:\Matisse\bin`, select `MatisseNet.dll`, click Open, and click OK.

Alternatively, you can reference the library from the command line by entering `csc /lib:LOCATION OF MatisseNet.dll /reference:MatisseNet.dll [/target:library] [/out:somename] ApplicationName`, where `ApplicationName` is your .NET program `.cs` or `.vb` source file.

3. Add each of the generated class files to the project (right-click on the project icon and select Add / Add Existing Item).

9.3 Generate Schema Documentation

For C# only, you may generate HTML documentation for the schema using NDoc, a free utility downloadable from ndoc.sourceforge.net.

1. In Visual Studio .NET, right-click on the project, select Properties, open the Configuration Properties folder, click Build, set the XML Documentation File property to `Schema.xml`, and click OK.
2. Compile the solution.
3. Click Add, select your compiled application as the Assembly Filename and `Schema.xml` as the XML Doc Filename, and click OK.
4. Click Add again, select `MatisseNet.dll` as the Assembly Filename and `Schema.xml` as the XML Doc Filename, and click OK.
5. Set the NDOC options as appropriate, then select Documentation / Build.

10 Code Generation and Data Classes

10.1 Code Generation

As explained in [9.1 Code Generation Using mt_dnom](#), generating classes to access Matisse database is the first step you will do when you build a new application.

The `mt_dnom` utility accesses the database and generates stub classes for all the classes defined in the database. Each stub class contains properties and methods that allow you to access the database for searching, reading, updating, or deleting objects. The generated properties and methods are listed in the next table.

Category	Name	Description
Constructor	<i>Class</i>	Default Constructor
Properties	<i>Attribute</i>	Gets and sets attribute value
	<i>Relationship</i>	Gets and sets successor objects for a relationship
	<i>RelationshipSize</i>	Gets the number of successor objects for a relationship
Methods	<i>GetInstancesNumber</i>	Returns the number of instances of the class and its all subclasses
	<i>GetOwnInstancesNumber</i>	Returns the number of instances of the class only (excluding its subclasses)
	<i>InstancesEnumerator</i>	Returns an enumerator to iterate through all the instances of the class and its all subclasses
	<i>OwnInstancesEnumerator</i>	Returns an enumerator to iterate through all the instances of the class only (excluding its subclasses)
	<i>RelationshipEnumerator</i>	Returns an enumerator to iterate through all the successor objects for a relationship
	<i>PrependRelationship</i> <i>AppendRelationship</i> <i>RemoveRelationship</i> <i>ClearRelationship</i>	Manages successor objects for a relationship: - Add an object (or objects) at the beginning of a relationship - Add an object (or objects) at the end of a relationship - Remove an object (or objects) from a relationship - Remove all the successor objects from a relationship
	<i>RemoveAttribute</i>	The attribute's value becomes "Unspecified" or falls back to the default value if there is one defined for the attribute

Category	Name	Description
	LookupIndex LookupEntryPointDictionary	Searches for an object with the specified key value(s) using the index or the entry-point dictionary. Returns the object if one object is matched. Returns null if no object is matched or more than one object are matched.
	IndexEnumerator EntryPointDictionaryEnumerator	Returns an enumerator for iteration over a collection of objects that match the specified key value(s) using the index or the entry-point dictionary.
	SQL method	Calls the SQL instance method defined in the database. Note that SQL static methods can be called using CALL statement with the ExecuteScalar method in ADO.NET

Note that the *italic* part of names above are replaced by the real name, for example, `firstName` for *Attribute* in class `Person`.

Executing SQL method

While SQL static methods in the database server can be called using a `CALL` statement with ADO.NET, SQL instance methods can be directly invoked in the .NET application without using a SQL statement. Here is an example of an SQL instance method:

```
CREATE METHOD CalculateBonus(rate DOUBLE)
RETURNS NUMERIC
FOR Manager
BEGIN
  DECLARE result NUMERIC;
  -- calculate the bonus for the manager
  RETURN result;
END;
```

After you generate stub classes, you will see a method definition in the `Manager` class as following:

```
public decimal CalculateBonus(double arg1) {
  ...
}
```

Then, in your program you can call the `CalculateBonus` SQL method with the next code:

```
Manager mgr;
mgr = ...; // retrieve a Manager object from the database
decimal bonus = mgr.CalculateBonus(0.1);
```

10.2 Data Classes

You may need to copy Matisse property values into application classes that are independent from the Matisse persistence layer. It can be used for the following purposes,

1. Convert objects using serialization into a format that is transmittable over the network in order to transfer the data to a remote application as a Web service.

- Put all the business logic in the application layer and keep the layer completely independent from the persistence layer. Copy the data back and forth between the application layer and persistence layer.

The Matisse .NET binding provides the capability to generate “Data Classes” that can be used for these purposes, in addition to the standard stub classes generated by the `mt_dnom` utility. Data classes share with the stub classes all the properties defined in the database schema, but do not include any database access methods. See [9.1 Code Generation Using `mt_dnom`](#) for the `mt_dnom` utility.

Generating Data Classes

Use the `mt_dnom` utility with the “-adc” option followed by a namespace in which the data classes will be defined. The namespace is mandatory since the data classes will be defined using the same name as the ones for stub classes. For example, the command

```
> mt_dnom -d mydb stubgen -lang C# -adc MyCompany.MyProject.MyBusinessLogic
```

will generate three files `Person.cs`, `Employee.cs`, and `Manager.cs` in the current working directory, and three other files `Person.cs`, `Employee.cs`, and `Manager.cs` in its subdirectory `DataClasses`, assuming your database contains these three classes. The three files in the directory `DataClasses` are data classes. The generated stub classes will include some additional methods to copy data, which will be explained in the next section.

When you need to update the database schema, you first upload the schema into the database using UML (Rational Rose), SQL DDL, or ODL (Object Definition Language). Then rerun the `mt_dnom` utility in the same directory. It will update the files for both stub classes and data classes while preserving all the user-added code.

10.3 Copying Data

Copying Attributes

When you copy the data of a persistent object, i.e., an instance of `MtObject` or its subclasses, use the `ToDataObject` method defined in the stub classes. This method creates a new instance of a data class, called data object, and copies all the Matisse attribute values from the persistent object to the data object.

```
// Execute the query "SELECT REF(m) FROM Manager m WHERE ..." and
// get the ADO.NET reader object.
// Then, get a Manager object.
Manager pMgr = (Manager) aDataReader.GetObject(0);

// Create a new data object for Manager and copy all the attribute values
MyProject.MyBusinessLogic.Manager dMgr =
    (MyProject.MyBusinessLogic.Manager) pMgr.ToDataObject();
```

The generated `ToDataObject` method in each stub class consists of three parts: creation of a new data object, copying of attribute values (the `CopyAttributesToDataObj` method), and copying of relationships (the `CopyRelationshipsToDataObj` method). The default `CopyAttributesToDataObj` method provided copies all the attribute values, while the default `CopyRelationshipsToDataObj` method does not copy any relationship values.

Copying Relationships

Now, suppose that you need to copy a `Manager` object with its assistant object (the relationship assistant defined in the `Manager` class). To this end, you will override the default `CopyRelationshipsToDataObj` method in the `Manager` stub class (not the data class).

```
public override void CopyRelationshipsToDataObj(object obj)
{
    // Cast the object to the specific data class.
    MyProject.BusinessLogic.Manager dataObj =
    (MyProject.BusinessLogic.Manager) obj;

    // Base classes may also define the method.
    base.CopyRelationshipsToDataObj(dataObj);

    // Create a new data object for assistant and assign it to the data
    object.
    dataObj.Assistant =
        (MyProject.BusinessLogic.Employee) this.Assistant.ToDataObject();
}
```

NOTE: When you copy relationships, be aware of the next common traps.

(1) You may end up copying almost the entire database content if all the objects are connected to each other and you copy all the relationships in these objects. For example, if you have a tree shaped data structure in the database, and you define the `CopyRelationshipsToDataObj` method so it copies all the sub-nodes of each node, then calling

```
aRootNode.ToDataObject();
```

will copy the entire tree.

(2) You may fall into an infinite loop. Suppose that you have written the `CopyRelationshipsToDataObj` method for `Person` so it copies the spouse relationship. If your code is simply like this:

```
dataObj.Spouse =
    (MyProject.BusinessLogic.Person)
this.Spouse.ToDataObject();
the execution will fall into an infinite loop.
```

10.4 Storing Data Object's Updates into the Database

After you create a data object and update the object, you may need to store the update back into the database. Since the property updates that you need to save depend on your application, the `mt_dnom` utility does not generate any code to store the data object's updates back into the database. But, here is a guide line for how to do it.

1. Define a method in the stub class to save updates, e.g., `Update`, with a data object argument. For example, to update the `comment` and `age` properties in the `Person` class, write the code in `Person.cs` like:

```
public void Update(object obj) {
    MyProject.BusinessLogic.Person dataObj =
    (MyProject.BusinessLogic.Person) obj;
    this.Comment = dataObj.Comment;
    this.Age = dataObj.Age;
}
```

2. In your application, you may retrieve the persistent object corresponding to the data object using SQL, the index lookup APIs, or Entrypoint lookup APIs. Instead of retrieving it, you may keep the persistent object. Then, call the Update method to store the update made to the data object. For example,

```
Person p;
MyProject.BusinessLogic.Person dataObj;

// Retrieve the persistent object that corresponds to the data object using
// the index API
p = Person.LookupPersonName(db, dataObj.LastName, dataObj.FirstName);

// Store the update
p.Update(dataObj);
```

10.5 The mt_dnom Utility

The usage of the `mt_dnom` utility is as follows:

```
$ mt_dnom
Matisse .NET Object Manager x32 Version 8.2.1 (32-bit Edition) - Nov 25 2008.
(c) Copyright 1992-2008 Matisse Software Inc. All rights reserved.
Usage:
  Generate Stubs:
    mt_dnom -d [user:]database[@host[:port]] [-p] stubgen [-lang C# | VB] [-n
<namespace>] [-adc <namespace>] [-[no]psm]
      -d [user:]database[@host] : the database to be accessed.
      -p                          : allows the user to authenticate with
                                  a username/password.
                                  - if the '-p' option is used, it will
                                  be assumed that the current system user
                                  is known from the database, but a
                                  password will be asked.
                                  - if the '-p' option is not used, Matisse
                                  <user> is used for user name, and a
                                  password will be asked.
                                  - if the user is not defined and
                                  the '-p' option is not used, it will
                                  be assumed that the current system user
                                  is known from the database, and does not
                                  need password.
    stubgen [-lang C# | VB] [-n <namespace>] [-adc <namespace>] [-[no]psm]
      -lang                          : generate C# or VB files from the database
                                  schema. The default is C#
      -n <namespace>                  : define the generated classes in the
```

```
                                specified namespace
-[no]psm                        : generate .NET methods mapping Persistent
                                SQL methods. The default is -psm
-adc <namespace>               : generate ADO Data Classes in addition
                                to stub classes
```

Stub classes are generated in the directory where the `mt_dnom` utility is executed. Data classes are generated in the `ADODataClasses` subdirectory in the directory where the `mt_dnom` utility is executed.

When you update the database schema, you need to run the `mt_dnom` utility again in the same directory to update the C# or VB.NET class files. The `mt_dnom` utility updates the files while preserving the part you added.

11 Generating Class Stubs with a CodeDOM Provider

The Matisse Class Stubs generator supports any .NET programming language that includes a .NET CodeDOM provider. This feature allows you to generate Matisse Stub Class to manipulate your persistent data from the .NET programming language of your choice.

11.1 Building your Class Stub Generator

The `MtClassStubGenerator` class generates the Class Stub source code corresponding to the Matisse database schema classes. Combined with a `CodeDomProvider`, the Matisse Class Stubs generator can generate the source code for any .NET programming language.

```
db.Open();
db.StartVersionAccess();

MtClassStubGenerator engine = new MtClassStubGenerator(db, codeProvider,
nopsm, utilityDesc);
languageName = engine.LanguageName;
codeSnippet = engine.GetNamespaceConnectCodeSnippet(nmspace, assemblyName);

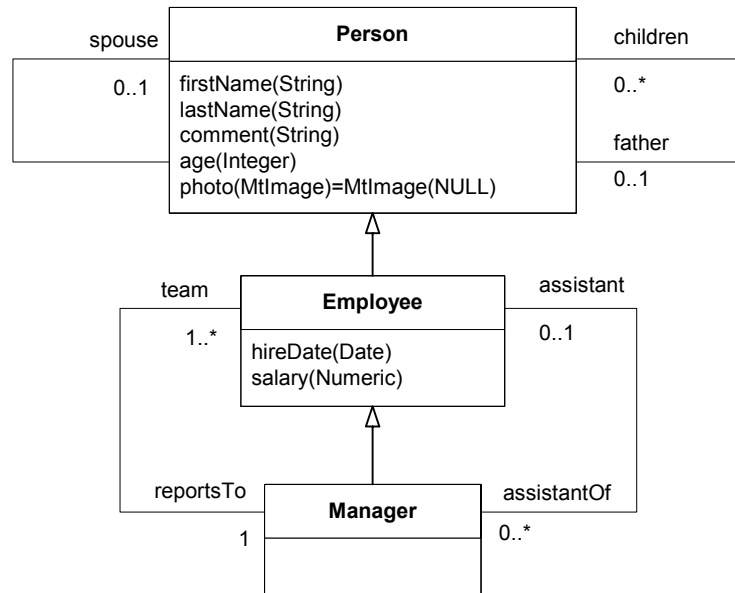
foreach (MtClass cls in MtClass.GetClass(db).InstancesEnumerator())
{
    if (!cls.IsPredefined())
    {
        engine.GenerateClassStub(cls, nmspace, (null != adc_nmspace),
adc_nmspace);
        if (null != adc_nmspace)
        {
            engine.GenerateDataClassStub(cls, DATA_CLASS_DIR_NAME, adc_nmspace);
        }
    }
}

db.EndVersionAccess();
db.Close();
```

11.2 Example

The StubGen C# project illustrates this new feature.

Appendix A: Example.odl Schema



```

interface Person : persistent
{
  attribute String firstName;
  attribute String lastName;
  attribute String Nullable comment;
  attribute Integer Nullable age;
  attribute Image Nullable photo = NULL;
  relationship Person spouse[0,1] inverse Person::spouse;
  readonly relationship Person father[0,1] inverse Person::children;
  relationship List<Person> children inverse Person::father;
  mt_index personName
    criteria {person::lastName MT_ASCEND 16},
             {person::firstName MT_ASCEND 16};
  mt_entry_point_dictionary commentDict entry_point_of comment
    make_entry_function "make-full-text-entry";
};

interface Employee : Person : persistent
{
  attribute Date hireDate;
  attribute Numeric salary;
  readonly relationship List<Manager> assistantOf inverse Manager::assistant;
  relationship Manager reportsTo inverse Manager::team;
};

interface Manager : Employee : persistent
{
  relationship List<Employee> team[1,-1] inverse Employee::reportsTo;
  relationship Employee assistant[0,1] inverse Employee::assistantOf;
};

```

Appendix B: Managing a Database Schema with Object APIs

While you can access or update the database schema with SQL DDL statements in your .NET application, you can achieve the same with the .NET object APIs for Matisse. Here is a sample program to get a descriptor object for a Matisse class and add/remove an attribute to/from the class:

```
MtDatabase dbcon;
dbcon = new MtDatabase (...);

// Set the DATA_ACCESS_MODE option so the transaction will be
// opened as "schema definition" mode.
// This is required only when you update the database schema.
dbcon.SetOption(MtDatabase.MtConnectionOption.DATA_ACCESS_MODE,
    (int) MtDatabase.MtDataAccessMode.DATA_DEFINITION);
dbcon.Open();
dbcon.BeginTransaction();

// Get the Manager class from the database
MtClass aClass = MtClass.Get (dbcon, "Manager");

// Add a new attribute to the class.
// The new attribute is of type Integer with the default value 0
MtAttribute newAttr = new MtAttribute (dbcon, "MgrRank",
MtType.MtBasicType.INTEGER, 0);
aClass.AppendMtAttributes (newAttr);

dbcon.Commit();

dbcon.BeginTransaction();

// Remove the new attribute from the class
MtAttribute anAttr = MtAttribute.Get (dbcon, "MgrRank", aClass);
anAttr.Remove();

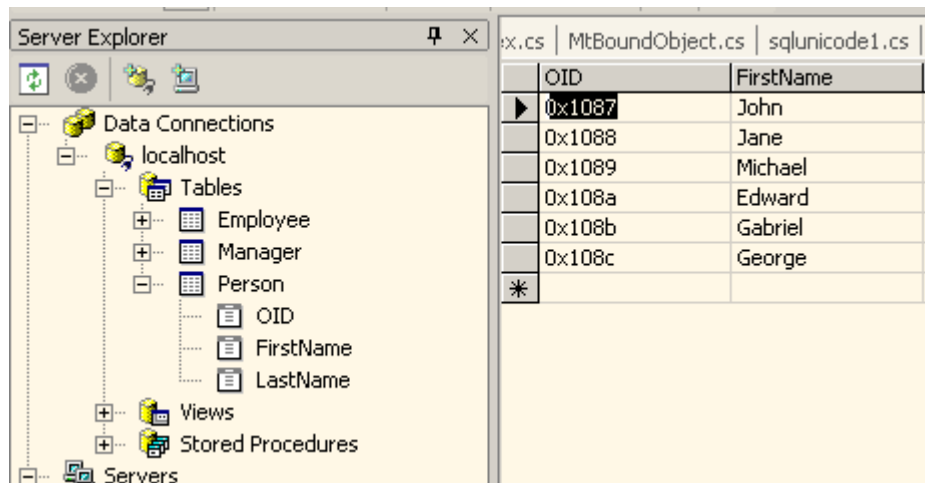
dbcon.Commit();
```

All the APIs for these classes are described in the reference manual, which is installed with the .NET binding.

Appendix C: Browsing Database Objects with Visual Studio .NET

You can browse objects in a Matisse database with Visual Studio .NET using the Matisse ODBC driver.

1. Install the Matisse ODBC driver of the appropriate version.
2. Define a data source with a name to access your database.
3. In Visual Studio .NET, open the “Server Explorer” tab, right-click on “Data Connections”, and select “Add Connection ...”.
4. In the window that just opened, select the “Provider” tab and choose “Microsoft OLE DB Provider for ODBC Drivers”, and click “Next>>” button.
5. In the “Connection” tab, specify the data source by choosing in the scroll list the item with the data source name that you created.
6. Then, click OK to add a connection to your database in the “Server Explorer” tab.
7. Now, you can browse the database.



Index

B

BeginTransaction 15

C

CALL 12

Command 7, 11

CommandType 11

Connection 15

D

Data Class 35

Data Types 15

DataAdapter 13

DataReader 8

DataRelation 14

DataSet 13

E

ExecuteNonQuery 7

ExecuteReader 7

ExecuteScalar 7, 10, 12

F

Fill 13

G

GetChildRows 14

GetObject 9

I

IsNull 13

M

MtCommand 7

MtDatabase 7

P

Parameter 15

Parameters 10, 11

R

Read 8

REF 9

Relationship 14

S

SelectCommand 13

Stored Methods 11

StoredProcedure 11

stub classes 31

T

ToDataObject 35

Transaction 15

Transactions 15