Matisse® .NET Programmer's Guide

May 2013

Matisse .NET Programmer's Guide

PDF generated 30 April 2013

Content

1 Introduction

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*.

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.

Before Running the Examples

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

- Install Matisse 9.x or higher.
- Install Visual Studio 2008 with .NET Framework 3.5 or higher

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:

- Matisse.Data.**MtDatabase** - manages the connection to Matisse database
- Matisse.Data.**MtCommand** - executes SQL statements or stored methods
- Matisse.Data.**MtDataReader** - retrieves the data (values or objects) of executed command
- 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.

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.

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.

Defining the Session Namespace

After you open a connection to a Matisse database, you set the SQL current namespace to a namespace path so you can access the schema objects without their full qualified names.

```
// Set the SQL CURRENT_NAMESPACE to 'Examples.Csharp.ADO' so there is
// no need to use the full qualified names to access the schema objects
dbcon.SqlCurrentNamespace = "Examples.Csharp.ADO";
```

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.

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 2.0.1 Typed accessor methods of DataReader

Matisse data type	Accessor method of DataReader	Return type
MT_BYTE	GetByte	byte
MT_SHORT	GetInt16	short
MT_INTEGER	GetInt32	int
MT_LONG	GetInt64	long
MT_FLOAT	GetFloat	float
MT_DOUBLE	GetDouble	double
MT_NUMERIC	GetDecimal	decimal
MT_STRING (ASCII)	GetString	string
MT_STRING (Unicode)	GetUnicode	string
MT_CHAR	GetChar	char
MT_BOOLEAN	GetBoolean	bool
MT_DATE	GetDateTime, GetDate	DateTime
MT_TIMESTAMP	GetDateTime, GetTimestamp	DateTime
MT_INTERVAL	GetInterval	TimeSpan
MT_OID	GetObject	MtObject
MT_NULL		DBNull.Value
MT_ANY	GetValue *	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 *16 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 *Connecting with an 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 *Matisse Data Types*.

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();
```

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;
```

Filling a DataSet Using a DataAdapter

A `DataAdpater` 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 `Conection` 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.Write(((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.

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*.

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();
```

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 2.0.1 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_INTEGET_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 will explain how to manipulate object with 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 *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 *16 Building an Application from Scratch*.

Running ObjectsExample

This sample program creates objects from 2 classes (`Person` and `Employee`), lists all `Person` objects (which includes both objects, since `Employee` is a subclass of `Person`), deletes objects, then lists all `Person` objects again to show the deletion. Note that because `FirstName` and `LastName` are not nullable, they *must* be set when creating an object.

Follow the instructions below 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 -f 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. In a command-line window, change to the `ObjectsExample bin` directory and run the application:

```
> ObjectsExample localhost example
```

Creating Objects

This section illustrates the creation of objects. The stubclass provides a default constructor which is the base constructor for creating persistent objects.

```
public Person(MtDatabase db) :
```

```
            base(GetClass(db)) {
    }

// Create a new Person object (instance of class Person)
Person p = new Person(db);
p.FirstName = "John";
p.LastName = "Smith";
p.Age = 42;
PostalAddress a = new PostalAddress(db);
a.City = "Portland";
a.PostalCode = "97201";
p.Address = a;
Console.WriteLine("Person John Smith created.");

// Create a new Employee object
Employee e = new Employee(db);
e.FirstName = "Jane";
e.LastName = "Jones";
// Age is nullable we can leave it unset
e.HireDate = DateTime.Now;
e.Salary = new Decimal(85000.00);
Console.WriteLine("Employee Jane Jones created.");
```

If your application need to create a large number of objects all at once, we recommend that you use the `Preallocate()` method defined on `MtDatabase` which provide a substantial performance optimization.

```
db.StartTransaction();

// Optimize the objects loading
// Preallocate OIDs so objects can be created in the client workspace
// without requesting any further information from the server
db.Preallocate(DEFAULT_ALLOCATOR_CNT);

for (int i = 1; i <= 100; i++)
{
    // Create a new Employee object
    Employee e = new Employee(db);

    String fname = fNameSample[sampleSeq.Next(MAX_SAMPLES)];
    String lname = lNameSample[sampleSeq.Next(MAX_SAMPLES)];
    e.FirstName = fname;
    e.LastName = lname;
    e.HireDate = DateTime.Now;
    e.Salary = new Decimal(salarySample[sampleSeq.Next(MAX_SAMPLES)]);
    PostalAddress a = new PostalAddress(db);
    int addrIdx = sampleSeq.Next(MAX_SAMPLES);
    a.City = addressSample[addrIdx][0];
    a.PostalCode = addressSample[addrIdx][1];
    e.Address = a;
    Console.WriteLine("Employee " + i + " " + fname + " " + lname + "
created.");
    if (i % OBJECT_PER_TRAN_CNT == 0)
```

```
    {
        db.Commit();
        db.StartTransaction();
    }
    // check the remaining number of preallocated objects.
    if (db.NumPreallocated() < 2)
    {
        db.Preallocate(DEFAULT_ALLOCATOR_CNT);
    }
}

if (db.IsTransactionInProgress())
    db.Commit();
```

Listing Objects

This section illustrates the enumeration of objects from a class. The `InstancesEnumerator()` static method defined on a generated stubclass allows you to enumerate the instances of this class and its subclasses. The `GetInstancesNumber()` method returns the number of instances of this class.

```
// List all Person objects
Console.WriteLine("\n" + Person.GetInstancesNumber(db) +
                  " Person(s) in the database.");
Console.WriteLine("\n" + PostalAddress.GetInstancesNumber(db) +
                  " Address(s) in the database.");
foreach (Person x in Person.InstancesEnumerator(db))
{
  Console.WriteLine("\t" + x.FirstName + " " + x.LastName +
                "  from " + (x.Address != null ? x.Address.City : "???") +
                "  is a " + x.MtClass.MtName);
}
```

The `OwnInstancesEnumerator()` static method allows you to enumerate the own instances of a class (excluding its subclasses). The `GetOwnInstancesNumber()` method returns the number of instances of a class (excluding its subclasses).

```
// List all Person objects (excluding Employee sub-class)
Console.WriteLine("\n" + Person.GetOwnInstancesNumber(db) +
                  " Person(s) (excluding subclasses) in the database.");

MtObjectEnumerator<Person> iter = Person.OwnInstancesEnumerator(db);

foreach (Person x in iter)
{
  Console.WriteLine("\t" + x.FirstName + " " + x.LastName +
                "  from " + (x.Address != null ? x.Address.City : "???") +
                "  is a " + x.MtClass.MtName);
}
```

Deleting Objects

This section illustrates the removal of objects. The `Remove()` method delete an object.

```
// Remove created objects
Person p;
...
// NOTE: does not remove the object sub-parts
p.Remove();
```

To remove an object and its sub-parts, you need to override the `DeepRemove()` method in the stubclass to meet your application needs. For example the implementation of `DeepRemove()` in the `Person` class that contains a reference to a `PostalAddress` object is as follows:

```
public override void DeepRemove()
{
    if (Address != null)
        Address.DeepRemove();
    base.DeepRemove();
}

Person p;
...
p.DeepRemove();
```

The `RemoveAllInstances()` method defined on `MtClass` delete all the instances of a class.

```
Person.GetClass(db).RemoveAllInstances();
```

Comparing Objects

This section illustrates how to compare objects. Persistent objects must be compared with the `Equal()` method. You can't compare persistent object with the == operator.

```
Person p1;
Person p2;
...
if(p1.Equals(p2))
    System.out.println("Same objects");
```

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` directory, and enter:

```
> ValuesExample localhost example
```

Setting and Getting Values

This section illustrates the set, update and read object property values. The stubclass provides a set and a get method for each property defined in the class.

```
Employee e = new Employee(db);

// Setting strings
e.Comment = "FirstName, LastName, Age, HireDate & Salary Set";
e.FirstName = "John";
e.LastName = "Jones";

// Setting numbers
e.Age = 42;

// Setting Date
e.HireDate = new DateTime(2008, 11, 8);

// Setting Numeric
e.Salary = new Decimal(7421.25);

// Getting
Console.WriteLine("\nComment: " + e.Comment);
Console.WriteLine("\t" + e.MtClass.MtName + ": " +
                    e.FirstName + " " +
                    e.LastName);
// suppresses output if no value set
if (!e.IsAgeNull())
Console.WriteLine("\t" + e.Age + " years old");
Console.WriteLine("\tNumber of dependents:  " +
                    e.Dependents);
Console.WriteLine("\tSalary:  $" + e.Salary);
Console.WriteLine("\tHiring Date:  " + e.HireDate.ToString("yyyy-MM-dd"));


// Setting NULL
Console.WriteLine("Setting Comment to null... ");
e.Comment = null;

Console.WriteLine("Setting Age to null... ");
e.SetNull(Employee.GetAgeAttribute(db));
// Since Age is Nullable. you can also write
e.Age = null;
```

Removing Values

This section illustrates the removal of object property values. Removing the value of an attribute will return the attribute to its default value.

```
    Employee e;

// Removing value returns attribute to default
e.RemoveAge();
if (!e.IsAgeNull())
    Console.WriteLine("\t" + e.Age + " years old");
else
    Console.WriteLine("\tAge: null" +
                        (e.IsAgeDefaultValue() ? " (default value)" : ""));
```

Streaming Values

This section illustrates the streaming of blob-type values (`MT_BYTES, MT_AUDIO, MT_IMAGE, MT_VIDEO`). The stubclass provides streaming methods (`SetPhotoElements()`, `GetPhotoElements()`) for each blob-type property defined in the class. It also provides a `PhotoSize` property to retrieve the blob size without reading it.

```
// Setting blobs

// set to 512 for demo purpose
// a few Mega-bytes would be more appropriate
// for real multimedia objects (audio, video, high-resolution photos)
int bufSize = 512;
int num, total;
byte[] buffer = new byte[bufSize];
try
{
    FileStream istream = new FileStream("..\\..\\..\\matisse.gif",
FileMode.Open, FileAccess.Read);

    // reset the stream
    e.SetPhotoElements(buffer, MtType.BEGIN_OFFSET, 0, true);
    do
    {
        //num = is.read(buffer, 0, bufSize);
        num = istream.Read(buffer, 0, bufSize);
        if (num > 0)
        {
            e.SetPhotoElements(buffer, MtType.CURRENT_OFFSET, num, false);
        }
    } while (num == bufSize);
    istream.Close();
}
catch (IOException ex)
{
```

```
    Console.WriteLine(ex.StackTrace);
}
Console.WriteLine("Image of " + e.PhotoSize + " bytes stored.");

Console.WriteLine("Streaming an image out...");
// Getting blobs (save value of e.Photo as out.gif in the
// program directory)
total = 0;
try
{
    FileStream ostream = new FileStream("out.gif", FileMode.OpenOrCreate,
FileAccess.Write);
    // reset the stream
    e.GetPhotoElements(buffer, MtType.BEGIN_OFFSET, 0);
    do
    {
        num = e.GetPhotoElements(buffer, MtType.CURRENT_OFFSET, bufSize);
        if (num > 0)
        {
            ostream.Write(buffer, 0, num);
        }
        total += num;
    } while (num == bufSize);
    ostream.Close();
}
catch (IOException ex)
{
    Console.WriteLine(ex.StackTrace);
}
Console.WriteLine("Image of " + total + " bytes read.");
```

4 Working with Relationships

One of the major 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. Relationship navigation is as easy as accessing an object property. There is no needs for executing a SQL query with joins nor using `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*.

Running RelationshipsExample

`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 instructions below.

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 -f 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. In a command-line window, change to the `RelationshipsExample bin` directory and run the application:

   ```
   > RelationshipsExample localhost example
   ```

Setting and Getting Relationship Elements

This section illustrates the set, update and get object relationship values. The stubclass provides a set and a get method for each relationship defined in the class.

```
Manager m1 = new Manager(db);
...
// Set a relationship
// Need to report to someone since the relationship
// cardinality minimum is set to 1
m1.ReportsTo = m1;

Employee e = new Employee(db);
...
// Set a relationship
e.ReportsTo = m1;

// Set a relationship
```

```
m1.Assistant = e;

Person c1 = new Person(db);
...
Person c2 = new Person(db);
...
// Set successors
m2.Children = new Person[] { c1, c2 };

// Get all successors

// assistantOf is automatically updated
Manager[] assistants = e.AssistantOf;

// father is automatically updated
Person f = c1.Father;
```

Adding and Removing Relationship Elements

This section illustrates the adding and removing of relationship elements. The stubclass provides a append, a remove and a clear method for each relationship defined in the class.

```
Person c3 = new Person(db);
...

// add successors
m2.AppendChildren(new Person[] { c3 });
...
// removing successors (this only breaks links, it does not
// remove objects)
m2.RemoveChildren(new Person[] { c2 });

// clearing all successors (this only breaks links, it does
// not remove objects)
m2.ClearChildren();
```

Listing Relationship Elements

This section illustrates the listing of relationship elements for one-to-many relationships. The stubclass provides an iterator method for each one-to-many relationship defined in the class.

```
// Iterate when the relationship is large is always more efficient
foreach (Person p in m2.ChildrenEnumerator())
{
    Console.Write(" " + p.FirstName);
}
```

Counting Relationship Elements

This section illustrates the counting of relationship elements for one-to-many relationships. The stubclass provides an get size method for each one-to-many relationship defined in the class.

```
// Get the relationship size without loading the .NET objects
// which is the fast way to get the size
long childrenCnt = m2.ChildrenSize;

Console.WriteLine("\t" + m2.FirstName + " has " + childrenCnt + " children");

// an alternative to get the relationship size
// but the .NET objects are loaded before you can get the count
childrenCnt = m2.Children.Length;
```

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 stubclass defines both lookup methods and iterator methods for each index defined on the class.

For example, the class `Person` which has an index `personName` defined on the two attributes `firstName` and `lastName`, generates several methods:

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

The lookup 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.

```
long GetPersonNameObjectNumber(MtDatabase db, string lastName, string
firstName, MtClass filterClass)
```

The GetObjectNumber method returns the count of `Person` objects matching the criteria. `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.

```
MtObjectEnumerator<Person> PersonNameEnumerator(MtDatabase db,
                                                string fromLastName,
                                                string fromFirstName,
                                                string toLastName,
                                                string toFirstName)
MtObjectEnumerator<Person> PersonNameEnumerator(MtDatabase db,
                                                string fromLastName,
                                                string fromFirstName,
                                                string toLastName,
                                                string toFirstName,
                                                MtClass filterClass,
                                                MtIndex.Direction direction,
                                                int numObjPerBuffer)
```

The enumerator 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 following Enumerator method arguments offer 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.

Additional index-related methods such as `GetIndexEntriesNumber` can be accessed directly from the `MtIndex` object.

```
long count = Person.GetPersonNameIndex(db).GetIndexEntriesNumber();
```

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` 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.

Index Lookup

This section illustrates retrieving objects from an index. The stubclass provides a lookup and a iterator method for each index defined on the class.

```
// the lookup function returns null to represent no match
Person found = Person.LookupPersonName(db, lastName, firstName);

// open an iterator for a specific range
string fromFirstName = "Fred";
string toFirstName = "John";
string fromLastName = "Jones";
string toLastName = "Murray";
Console.WriteLine("\nLookup from \"" +
                  fromFirstName + " " + fromLastName + "\" to \"" +
                  toFirstName + " " + toLastName + "\"");

foreach (Person p in Person.PersonNameEnumerator(db, fromLastName,
                                                 fromFirstName,
                                                 toLastName, toFirstName))
{
    Console.WriteLine("  " + p.FirstName + " " + p.LastName);
}
```

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 start or end.

```
e = Person.PersonNameEnumerator(db, null, null, "Murray", "John"));
```

Index Lookup Count

This section illustrates retrieving the object count for a matching index key. The `GetObjectNumber`() method is defined on the stub class returns the object count for a matching index key.

```
long count = Person.GetPersonNameObjectNumber(db, lastName, firstName, null);
Console.WriteLine(count + " objects retrieved");
```

Index Entries Count

This section illustrates retrieving the number of entries in an index. The `GetIndexEntriesNumber`() method is defined on the `MtIndex` class.

```
long count = Person.GetPersonNameIndex(db).GetIndexEntriesNumber();
Console.WriteLine(count + " entries in the index");
```

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` stub class which has an full-text entry-point dictionary `commentDict` defined on the attribute `comment`, will generate several methods:

```
Person LookupCommentDict(MtDatabase db, string ep)
Person LookupCommentDict(MtDatabase db, string ep, MtClass filterClass)
```

The lookup methods return 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. `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.

```
Person[] LookupObjectsCommentDict(MtDatabase db, string ep)
Person[] LookupObjectsCommentDict(MtDatabase db, string ep, MtClass
filterClass)
```

The lookupObjects methods return an array of `Person` objects that match the criteria.

```
long GetCommentDictObjectNumber(MtDatabase db, string ep)
long GetCommentDictObjectNumber(MtDatabase db, string ep, MtClass filterClass)
```

The GetObjectNumber methods return the count of `Person` objects matching the criteria.

```
MtObjectEnumerator<Person> CommentDictEnumerator(MtDatabase db, string ep)
MtObjectEnumerator<Person> CommentDictEnumerator(MtDatabase db, string ep,
MtClass filterClass)
MtObjectEnumerator<Person> CommentDictEnumerator(MtDatabase db, string ep, int
numObjPerBuffer)
MtObjectEnumerator<Person> CommentDictEnumerator(MtDatabase db, string ep,
MtClass filterClass, int numObjPerBuffer)
```

The Enumerator methods allow you to iterate through all the `Person` objects that match the criteria. `NumObjPerBuffer` is the number of matched objects transferred from the server to the client by each call.

> **NOTE:** The methods with the `FilterClass` argument are generated only if the class has sub-classes.

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, the example retrieves the `Person` objects in the database with `Comments` fields containing a specified character string.

To run the application, open a command-line window, change to the EPDictExample directory and go to the appropriate `bin` directory, and enter:

```
> EPDictExample localhost example search_string
```

Entry-Point Dictionary Lookup

This section illustrates retrieving objects from an entry-point dictionary. The stubclass provides both lookup methods and enumerator methods for each entry-point dictionary defined on the class.

```
// the lookup function returns null to represent no match
// if more than one match an exception is raised
Person found = Person.LookupCommentDict(db, searchstring);

// the LookupObjects method returns an array of matching objects
Person[] founds = Person.LookupObjectsCommentDict(db, searchstring);

long hits = 0;

// open an enumerator on the matching person objects
foreach (Person person in Person.CommentDictEnumerator(db, searchstring))
{
    Console.WriteLine("  " + person.FirstName + " " + person.LastName);
    hits++;
}
Console.WriteLine(hits + " Person(s) with 'comment' containing '" + searchstring
+ "'");
```

Entry-Point Dictionary Lookup Count

This section illustrates retrieving the object count for a matching entry-point key. The `GetCommentDictObjectNumber`() method defined on the stub class returns the object count for a matching entry-point key.

```
long count = Person.GetCommentDictObjectNumber(db, searchstring);
Console.WriteLine(count + " matching object(s) retrieved");
```

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.

Note that in this chapter there is no ODL file as you do not need to create an application schema.

Running 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 directory in each project, and run the command with two arguments <hostname> and <database name>.

Read Write Transaction

The following code connects to a database, starts a transaction, commits the transaction, and closes the connection.

```
MtDatabase db = new MtDatabase(args[0], args[1]);

db.Open();
db.StartTransaction();
```

```
// read/write access to the database

db.Commit();
db.Close();
```

Read-Only Access

The following code connects to a database in read-only mode, suitable for reporting.

```
MtDatabase db = new MtDatabase(args[0], args[1]);

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

// read-only access to the database

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

Version Access

The following code illustrates methods of accessing various versions of a database.

```
public static void ListVersions(MtDatabase db)
{
    foreach(string v in db.VersionEnumerator())
        Console.WriteLine("\t" + v);
}

MtDatabase db = new MtDatabase(args[0], args[1]);

db.Open();
db.StartTransaction();

Console.WriteLine("Version list before regular commit:");
ListVersions(db);
// read/write access to the database
db.Commit();

db.StartTransaction();
Console.WriteLine("Version list after regular commit:");
ListVersions(db);
// another read/write access
string verName = db.Commit("test"); // commit the transaction and make a
snapshot of a database so you can access this snapshot later
Console.WriteLine("Commit to version named: " + verName);

db.StartVersionAccess();
Console.WriteLine("Version list after named commit:");
```

```
    ListVersions(db);
    // read-only access on the latest version.
    db.EndVersionAccess();

    db.StartVersionAccess(verName);
    Console.WriteLine("Sucessful access within version: " + verName);
    // read-only access on a named version. It's not possible to
    // access a named version in read/write (transaction) mode.
    db.EndVersionAccess();

    db.Close();
```

Specific Options

This example shows how to enable the local client-server memory transport and to set or read various connection options and states.

```
using System;
using Matisse;

public class AdvancedConnect
{
    static MtDatabase db;

    public static void Main(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Need to specify <HOST> <DATABASE>");
            Environment.Exit(1);
        }
        try
        {
            db = new MtDatabase(args[0], args[1]);

            if (Environment.GetEnvironmentVariable("MT_MEM_TRANS") != null)
              db.SetOption(MtDatabase.ConnectionOption.MEMORY_TRANSPORT, MtDatabase.ON);

            if (Environment.GetEnvironmentVariable("MT_DATA_ACCESS") != null)
            {
                db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
                  Int32.Parse(Environment.GetEnvironmentVariable("MT_DATA_ACCESS")));
            }

            db.Open(Environment.GetEnvironmentVariable("dbuser"),
              Environment.GetEnvironmentVariable("dbpasswd"));

            Start(IsReadOnly());
            PrintState();

            End();

            db.Close();
        }
        catch (MtException mte)
```

```
    {
        Console.WriteLine("MtException : " + mte.Message);
    }
}

static void Start(bool readOnly)
{
    if (readOnly)
        db.StartVersionAccess();
    else
        db.StartTransaction();
}

static void End()
{
    if (db.IsVersionAccessInProgress())
        db.EndVersionAccess();
    else if (db.IsTransactionInProgress())
        db.Commit();
    else
        Console.WriteLine("No transaction/version access in progress");
}

static bool IsMemoryTransportOn()
{
    return db.GetOption(MtDatabase.ConnectionOption.TRANSPORT_TYPE) ==
        (int) MtDatabase.TransportType.MEM_TRANSPORT;
}

static bool IsReadOnly()
{
    return (db.GetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE) ==
        (int) MtDatabase.DataAccessMode.DATA_READONLY);
}

static void PrintState()
{
    if (!db.IsConnectionOpen())
    {
        Dbmsg("not connected");
    }
    else
    {
    if (db.IsTransactionInProgress())
        Dbmsg("read-write transaction underway");
    else if (db.IsVersionAccessInProgress())
        Dbmsg("read-only version access underway");
    else
        Dbmsg("no transaction underway");
    }
    Dbmsg("MEMORY_TRANSPORT is " + (IsMemoryTransportOn() ? "on" : "off"));

    Dbmsg("DATA_ACESS_MODE is " +
      (IsReadOnly()?"read only":"read write"));
}

static void Dbmsg(string msg)
{
    Console.WriteLine("database " + db.Name
```

```
            + " on server " + db.Host
            + ":  " + msg);
    }
}
```

More about MtDatabase

As illustrated by the previous sections, the `MtDatabase` class provides all the methods for database connections and transactions. The reference documentation for the `MtDatabase` class is included in the Matisse .NET Binding API documentation.

Connecting with an Object Factory

When your persistent classes are defined in a specific namespace 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 named `Schema` without any specific namespace, while the schema classes are defined in the `Examples.Csharp.Chap_3` namespace. In this case, you need to pass an `MtPackageObjectFactory` object as the additional argument for the `MtDatabase` constructor.

```
MtDatabase db = new MtDatabase(hostname, dbname,
    new MtPackageObjectFactory(",Schema",
                               "Examples.Csharp.Chap_3"));
```

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

<lang namespace>,<assembly name>

`<lang 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 `<lang namespace>`. For example, if the persistent classes are in the namespace `MyComp.Project1` in the assembly `proj1.dll and` the schema classes are defined in the `com.project1` namespace, then the constructor would look like

```
new MtPackageObjectFactory("MyComp.Project1,proj1", "com.project1")
```

For more information about the object factories, see the section *12 Working With Object Factories*.

8 Working with LINQ

This section demonstrates Matisse Language Integrated Query provider (LINQ). This example shows how to generate the LINQ Data Context class with the `mt_dnom` utility and how to query a database using .NET LINQ expressions.

Running LinqExample

This example creates a database with thousands of objects, then query them to illustrate Matisse LINQ provider. To run the sample program, follow the instructions below.

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 `LINQ` directory and load into the database the database schema `linqschema.odl`, which is an ODL (Object Definition Language) file.

   ```
   > mt_sdl -d example import --odl -f linqschema.odl

   Then load the dataset from the XML file:

    > mt_xml -d example import -f linqdata.xml
   ```

3. Open `LINQ.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `LinqExample` application.
4. In a command-line window, change to the appropriate `LinqExample bin` directory and run the application:

   ```
   > LinqExample localhost example
   ```

Generating the Data Context Class

The Linq Data Context class is generated in the directory where the `mt_dnom` utility is executed. By default, the Linq Data Context class name is set to the database name unless you specify its name with the `-linq` command line option. The Linq Data Context class name is generated in a file named `<Linq class>Linq`.

In the Linq example, the generated Linq Data Context class name is named `linqExample` and it was generated with the `mt_dnom` utility as follows:

```
 > mt_dnom -d example stubgen -lang C# -ln TestLinq.Schema -sn
Examples.Csharp.LINQ -linq linqExample
```

NOTE: The Linq Data Context class is not generated unless the `-linq` command line option is specified.

Using Matisse Linq Assembly

The Matisse Linq provider is not included in the `MatisseNet` assembly, but it is included in an additional library named `MatisseNet.Linq`.

Retrieving Objects

This example illustrates how to retrieve objects and navigate through relationships using a LINQ query expression.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);
// Enable Trace
mdb.Log = Console.Out;

// Query Expression
var projs = from p in mdb.Projects where p.Budget < 250 select p;

// Navigate through relationships
foreach (var proj in projs)
{
  Console.WriteLine("Project {0} is managed by EmpId {1}.",
                    proj.ProjectID, proj.ManagedBy.EmpId);
  foreach (var mbr in proj.Members)
  {
    Console.WriteLine("Project {0} has for team member {1}.",
                      proj.ProjectID, mbr.EmpId);
  }
}
```

NOTE: The `Log` property defined on `the MtEntityProvider` class enables to log the executed query statement and its parameter values.

Retrieving Views

This example illustrates how to retrieve views from objects using a LINQ query with navigation.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);

// Query Expression
var projview1 = from p in mdb.Projects where p.Budget < 250
                select new { p.ProjectID, p.Start, p.Finish };
```

```
// Query Expression
var projview2 = from p in mdb.Projects where p.Budget < 250
                select new { Project = p.ProjectID,
                             Manager = p.ManagedBy.EmpId,
                             City = p.ManagedBy.Address.City };
```

Navigating through Relationships

This example illustrates how to navigate across objects with relationship expressions.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);

// Query Expression with relationships
var emps3 = from e in mdb.Managers
            where e.Department.Location == "Portland"
            from m in e.DirectReports
            orderby e.LastName, m.Salary descending
            select new { MgrId = e.EmpId, MgrName = e.LastName,
                         ReportsName = m.LastName, ReportsSalary = m.Salary };
```

Navigating using Joins with class filtering

This example illustrates how to navigate across objects with join expressions.

```
// Query Expression with filtering
var mgrs2 = from d in mdb.Departments
            join e in mdb.Managers on mdb.Fkey(d.Employees).MtOid equals
e.MtOid
            where e.Expertise.Contains("Design") && e.Address.City ==
"Sevilla"
            from p in e.ManageProjects
            where p.Budget > 1000
            from m in p.Members
            orderby d.DepartmentName, e.LastName, p.ProjectID, m.LastName
            select new { Department = d.DepartmentName,
                         ManagerId = e.LastName, ProjectId = p.ProjectID,
                         Budget = p.Budget, MemberId = m.LastName };
```

NOTE: The `Fkey()` method defined on `theMtEntityProvider` class is a Matisse extension to build a join expression on the Matisse `OID` property.

Reporting with Group by

This example illustrates how to build a report with LINQ expression using group by statements.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);

// Query Expression Syntax
var stmt3 = from d in mdb.Departments
            from e in d.Employees
            group e by new { d.DepartmentName, e.MtClassName } into g
            select new { Department = g.Key.DepartmentName,
                         Position = g.Key.MtClassName,
                         Count = g.Count() };

// Query Expression Syntax
var stmt4 = from e in mdb.Employees
            group e by e.Department.DepartmentName into g
            orderby g.Key descending
            select new { Department = g.Key,
                         TotalSalary = g.Sum(e => e.Salary) };
```

Calling SQL Methods

This example illustrates how to execute SQL Methods in LINQ query statements.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);

// Static SQL Method Call
string[] bankNames = Bank.ListBankName(conn, false);

// Query Expression
var stmt1 = from d in mdb.Departments
            from e in d.Employees
            orderby d.DepartmentName, e.LastName
            select new { d.DepartmentName, e.LastName,
            BankName = mdb.Lelt(e.GetAccrualsName(bankNames)),
            Total = mdb.Lelt(e.GetAccrualsQuantity(bankNames)) };
```

NOTE: The `Lelt()` method defined on `theMtEntityProvider` class is a Matisse extension to explode a value of type list returned by a property or by a SQL method.

Compiling Queries

This example illustrates how to create compiled query statements.

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

// Get the database Entity Provider
TestLinq.Schema.LinqExample mdb = new TestLinq.Schema.LinqExample(conn);

// Query Expression
var fn = MtCompiledQuery.Compile((string deptName, string[] bankNames) =>
            from d in mdb.Departments
            from e in d.Employees
            where d.DepartmentName == deptName
            orderby d.DepartmentName, e.LastName
            select new
            {
            d.DepartmentName,
            e.LastName,
            BankName = mdb.Lelt(e.GetAccrualsName(bankNames)),
            Total = mdb.Lelt(e.GetAccrualsQuantity(bankNames))
            });

string[] banks = new string[] { "Overtime", "Vacation" };

mdb.Log = Console.Out;
// Execute query
var items = fn("Customer Care", banks).ToList();
```

9 Working with Class Reflection

This section illustrates Matisse Reflection mechanism. This example shows how to manipulate persistent objects without having to create the corresponding .NET stubclass with the `mt_dnom` utility. It also presents how to discover all the object properties.

Running ReflectionExample

This example creates several objects, then manipulates them to illustrate Matisse Reflection mechanism. To run the sample program, follow the instructions below.

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 `reflection` 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 -f examples.odl
```

3. Open `Reflection.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `ReflectionExample` application.
4. In a command-line window, change to the appropriate `ReflectionExample bin` directory and run the application:

```
> ReflectionExample localhost example
```

Creating Objects

This example shows how to create persistent objects without the corresponding .NET stubclass. The static method `Get()` defined on all Matisse Meta-Schema classes (i.e. `MtClass`, `MtAttribute`, etc.) allows you to access to the schema descriptor necessary to create objects. Each object is an instance of the `MtObject` base class. The `MtObject` class holds all the methods to update the object properties (attribute and relationships (i.e. `SetString()`, `SetSuccessors()`, etc.).

```
MtDatabase db = new MtDatabase(hostname, dbname, new MtCoreObjectFactory());

db.Open();
db.StartTransaction();

Console.WriteLine("Creating one Person...");
// Create a Person object
MtClass pClass = MtClass.Get(db, "Person");
MtAttribute fnAtt = MtAttribute.Get(db, "FirstName", pClass);
MtAttribute lnAtt = MtAttribute.Get(db, "LastName", pClass);
MtAttribute cgAtt = MtAttribute.Get(db, "collegeGrad", pClass);
MtObject p = new MtObject(pClass);
p.SetString(fnAtt, "John");
p.SetString(lnAtt, "Smith");
```

```
p.SetBoolean(cgAtt, false);

Console.WriteLine("Creating one Employee...");
// Create a Employee object
MtClass eClass = MtClass.Get(db, "Employee");
MtAttribute hdAtt = MtAttribute.Get(db, "hireDate", eClass);
MtAttribute slAtt = MtAttribute.Get(db, "salary", eClass);
MtObject e = new MtObject(eClass);
e.SetString(fnAtt, "James");
e.SetString(lnAtt, "Roberts");

e.SetDate(hdAtt, new DateTime(2009, 1, 6));
e.SetNumeric(slAtt, new Decimal(5123.25));
e.SetBoolean(cgAtt, true);

Console.WriteLine("Creating one Manager...");
// Create a Manager object
MtClass mClass = MtClass.Get(db, "Manager");
MtRelationship tmRshp = MtRelationship.Get(db, "team", mClass);
MtObject m = new MtObject(mClass);
m.SetString(fnAtt, "Andy");
m.SetString(lnAtt, "Brown");
m.SetDate(hdAtt, new DateTime(2008, 11, 8));
m.SetNumeric(slAtt, new Decimal(7421.25));
m.SetSuccessors(tmRshp, new MtObject[] { m, e });
m.SetBoolean(cgAtt, true);

db.Commit();
db.Close();
```

Listing Objects

This example shows how to list persistent objects without the corresponding .NET stubclass. The `InstancesEnumerator()` method defined on the `MtClass` object allows you to access all instances defined on the class.

```
MtDatabase db = new MtDatabase(hostname, dbname, new MtCoreObjectFactory());

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

MtClass pClass = MtClass.Get(db, "Person");
MtAttribute fnAtt = MtAttribute.Get(db, "FirstName", pClass);
MtAttribute lnAtt = MtAttribute.Get(db, "LastName", pClass);
MtAttribute cgAtt = MtAttribute.Get(db, "collegeGrad", pClass);

// List all objects
Console.WriteLine("\n" + pClass.GetInstancesNumber() +
                        " Person(s) in the database.");

// Retrieve the object from the previous transaction
foreach (MtObject p in pClass.InstancesEnumerator<MtObject>())
```

```
{
    Console.WriteLine("- " + p.MtClass.MtName + " #" + p.MtOid);

    Console.WriteLine("  " + p.GetString(fnAtt) + " " +
                             p.GetString(lnAtt) +
                             " collegeGrad=" + p.GetBoolean(cgAtt));
}

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

Working with Indexes

This example shows how to retrieve persistent objects from an index. The `MtIndex` class holds all the methods retrieves objects from an index key.

```
MtClass pClass = MtClass.Get(db, "Person");
MtAttribute fnAtt = MtAttribute.Get(db, "FirstName", pClass);
MtAttribute lnAtt = MtAttribute.Get(db, "LastName", pClass);

// Get the Index Descriptor object
MtIndex iClass = MtIndex.Get(db, "personName");

// Get the number of entries in the index
count = iClass.GetIndexEntriesNumber();
Console.WriteLine(count + " entries in the index.");

Console.WriteLine("Looking for: " + firstName + " " + lastName);

// lookup for the number of objects matching the key
Object[] key = new Object[] { lastName, firstName };
count = iClass.GetObjectNumber(key, null);
Console.WriteLine(count + " matching objects to be retrieved.");

if (count > 1)
{
    // More than one matching object
    // Retrieve them with an iterator
    foreach (MtObject p in iClass.Enumerator<MtObject>(key))
    {
        Console.WriteLine("  found " + p.GetString(fnAtt) + " " +
                                  p.GetString(lnAtt));
    }
}
else
{
    // At most 1 object
    // Retrieve the matching object with the lookup method
    MtObject p = iClass.Lookup(key);
    if (p != null)
    {
        Console.WriteLine("  found " + p.GetString(fnAtt) + " " +
```

```
                                p.GetString(lnAtt));
        }
        else
        {
            Console.WriteLine("  Nobody found");

    }
```

Working with Entry Point Dictionaries

This example shows how to retrieve persistent objects from an Entry Point Dictionary. The `MtEntryPointDictionary` class holds the methods to retrieve objects from a string key.

```
    MtClass pClass = MtClass.Get(db, "Person");
    MtAttribute fnAtt = MtAttribute.Get(db, "FirstName", pClass);
    MtAttribute lnAtt = MtAttribute.Get(db, "LastName", pClass);
    MtAttribute cgAtt = MtAttribute.Get(db, "collegeGrad", pClass);

    // Get the Index Descriptor object
    MtEntryPointDictionary epClass = MtEntryPointDictionary.Get(db,
    "collegeGradDict");

    Console.WriteLine("Looking for Persons with CollegeGrad=" + collegeGrad);

    // lookup for the number of objects matching the key
    long count = epClass.GetObjectNumber(collegeGrad, null);
    Console.WriteLine(count + " matching objects to be retrieved.");

    if (count > 1)
    {
        // More than one matching object
        // Retrieve them with an iterator
        foreach (MtObject p in epClass.Enumerator<MtObject>(collegeGrad))
        {
            Console.WriteLine("  found " + p.GetString(fnAtt) + " " +
                                    p.GetString(lnAtt) +
                                    " collegeGrad=" + p.GetBoolean(cgAtt));
        }
    }
    else
    {
        // At most 1 object
        // Retrieve the matching object with the lookup method
        MtObject p = epClass.Lookup(collegeGrad);
        if (p != null)
        {
            Console.WriteLine("  found " + p.GetString(fnAtt) + " " +
                                    p.GetString(lnAtt) +
                                    " collegeGrad=" + p.GetBoolean(cgAtt));
        }
        else
```

```
        {
            Console.WriteLine("  Nobody found");
        }
    }
```

Discovering Object Properties

This example shows how to list the properties directly from an object. The `MtObject` class holds the `AttributesEnumerator()` method, `RelationshipsEnumerator()` method and `InverseRelationshipsEnumerator()` method which enumerate the object properties.

```
    // Retrieve the object from the previous transaction
    foreach (MtObject p in pClass.InstancesEnumerator<MtObject>())
    {
        Console.WriteLine("- " + p.MtClass.MtName + " #" + p.MtOid);

        Console.WriteLine(" Attributes:");
        MtPropertyEnumerator<MtAttribute> propIter = p.AttributesEnumerator();
        String propName;
        MtType.BasicType propType, valType;
        String fmtVal;
        foreach (MtAttribute a in propIter)
        {
            propName = a.MtName;
            propType = a.GetMtType();
            valType = p.GetType(a);
            fmtVal = null;
            switch (valType)
            {
                case MtType.BasicType.DATE:
                    fmtVal = p.GetDate(a).ToString("yyyy-MM-dd");
                    break;
                case MtType.BasicType.NUMERIC:
                    fmtVal = p.GetNumeric(a).ToString();
                    break;
                case MtType.BasicType.NULL:
                    fmtVal = null;
                    break;
                default:
                    fmtVal = p.GetValue(a).ToString();
                    break;
            }
            Console.WriteLine("\t" + propName +
                                    " (" + MtType.ToString(propType) +
                              "):\t" + fmtVal + " (" + MtType.ToString(valType)
    + ")");
        }
        propIter.Close();

        Console.WriteLine(" Relationships:");
```

```
    MtPropertyEnumerator<MtRelationship> rshpIter =
p.RelationshipsEnumerator();
    foreach (MtRelationship r in rshpIter)
    {
        Console.WriteLine("\t" + r.MtName +
                          ":\t" + p.GetSuccessorSize(r) + " element(s)");
    }

    Console.WriteLine(" Inverse Relationships:");
    rshpIter = p.InverseRelationshipsEnumerator();
    foreach (MtRelationship r in rshpIter)
    {
        Console.WriteLine("\t" + r.MtName +
                          ":\t" + p.GetSuccessorSize(r) + " element(s)");
    }
}
```

Adding Classes

This example shows how to add a new class to the database schema. The connection needs to be open in the DDL (`MtDatabase.DataAccessMode.DATA_DEFINITION`) mode. Then you need to create instances of `MtClass`, `MtAttribute` and `MtRelationship` and connect them together.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Creating 'PostalAddress' class and linking it to
'Person'...");

// Create a new Class

// Create attributes
MtAttribute cAtt = new MtAttribute(db, "City", MtType.BasicType.STRING);
MtAttribute pcAtt = new MtAttribute(db, "PostalCode",
MtType.BasicType.STRING);

MtClass paClass = new MtClass(db, "PostalAddress", new MtAttribute[] { cAtt,
pcAtt }, null);

MtClass pClass = MtClass.Get(db, "Person");
MtRelationship adRshp = new MtRelationship(db, "Address", paClass, new int[]
{ 0, 1 });
pClass.AddMtRelationship(adRshp);

db.Commit();
```

Adding Attributes

This example shows how to add a new attribute to the database schema class. The connection needs to be open in the DDL (`MtDatabase.DataAccessMode.DATA_DEFINITION`) mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Adding 'Country' attribute to 'PostalAddress' class...");

// Create a new class attribute with a default value and with data
// type constaints
// NOTE: need a default value since the class which already exists may already
// have instances
MtAttribute cAtt = new MtAttribute(db, "Country", new MtType(db,
MtType.BasicType.STRING, MtType.CharEncoding.CODE_ASCII, 32), "USA");

// Set attribute not nullable so attribute can be indexed
cAtt.MtNullable = false;

// Retrieve the Class
MtClass paClass = MtClass.Get(db, "PostalAddress");

// add the newly created attribute to the class attribute list
paClass.AppendMtAttributes(cAtt);

db.Commit();
```

Adding Relationships

This example shows how to add a new relationship to the database schema class. The connection needs to be open in the DDL (`MtDatabase.DataAccessMode.DATA_DEFINITION`) mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Adding 'Friends' relationship to 'Person' class...\n");

MtClass pClass = MtClass.Get(db, "Person");

// Create a many-to-many relationship with self inverse (i.e. Friends,
Contacts, etc.)
// ODL definition: relationship Set<Person> Friends[0, -1] inverse
Person::Friends;
```

```
MtRelationship fRshp = new MtRelationship(db, "Friends", pClass, new int[] {
0, -1 });
fRshp.MtInverseRelationship = fRshp;

// uncomment the line below to get the List instead of a Set
// ODL definition:   relationship List<Person> Friends[0, -1] inverse
Person::Friends;
//fRshp.MtPreserveOrder = true;

// Establish a many-to-many relationship between the "Person" class and
// the "Person" class
pClass.AddMtRelationship(fRshp);

Console.WriteLine("Adding 'SpecialProjectTeam' relationship to 'Manager'
class...");
Console.WriteLine("and its inverse 'SPTeamMember' relationship to 'Employee'
class...\n");

MtClass mClass = MtClass.Get(db, "Manager");
MtClass eClass = MtClass.Get(db, "Employee");

// Create a one-to-many relationship between 2 different classes
// ODL definition:
//interface Manager : Employee : persistent {
//  relationship Set<Employee> SpecialProjectTeam[0, -1]
//    inverse Employee::SPTeamMember;
//};
//interface Employee : Person : persistent {
//  relationship Manager SPTeamMember[0, 1]
//    inverse Manager::SpecialProjectTeam;
//};
MtRelationship sptRshp = new MtRelationship(db, "SpecialProjectTeam",
eClass, new int[] { 0, -1 },"SPTeamMember", mClass, new int[] { 0, 1 });
// uncomment the line below to get the List instead of a Set
//sptRshp.MtPreserveOrder = true;

// Establish a one-to-many relationship between the "Manager" class and
// the "Employee" class
mClass.AddMtRelationship(sptRshp);
eClass.AddMtRelationship(sptRshp.MtInverseRelationship);


Console.WriteLine("Adding 'DeputyManager' relationship to 'Manager'
class...");
Console.WriteLine("and its inverse 'DeputyManagerOf' relationship to
'Employee' class...\n");

// Create a relationship between 2 different classes with a read-only inverse
// ODL definition:
//interface Manager : Employee : persistent {
//  relationship Employee DeputyManager[0, 1]
//    inverse Employee::DeputyManagerOf;
//};
//interface Employee : Person : persistent {
```

```
//  readonly relationship Manager DeputyManagerOf[0, 1]
//    inverse Manager::DeputyManager;
//};
MtRelationship dmRshp = new MtRelationship(db, "DeputyManager", eClass, new
int[] { 0, 1 },"DeputyManagerOf", mClass, new int[] { 0, 1 });

mClass.AddMtRelationship(dmRshp);
// uncomment the line below and it is no longer read-only
//eClass.AddMtRelationship(dmRshp.MtInverseRelationship);

db.Commit();
```

Adding Index

This example shows how to add a new index to the database schema. The connection needs to be open in the DDL (`MtDatabase.DataAccessMode.DATA_DEFINITION`) mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Creating 'CountryIdx' that indexes 'Country' on
'PostalAddress'...");

// Retrieve the class
MtClass paClass = MtClass.Get(db, "PostalAddress");

// Retrieve the attribute
MtAttribute cAtt = MtAttribute.Get(db, "Country", paClass);

// ODL definition: mt_index CountryIdx criteria {PostalAddress::Country
MT_DESCEND};
MtIndex cIdx = new MtIndex(db, "CountryIdx", paClass, new MtAttribute[] {
cAtt }, new int[] { (int)MtIndex.Ordering.DESCEND });

db.Commit();
```

Deleting Objects

This example shows how to delete persistent objects.The `MtObject` class holds `Remove()` and `DeepRemove()`. Note that on `MtObject` `DeepRemove()` does not execute any cascading delete but only calls `Remove()`.

```
// Retrieve the object from the previous transaction
foreach (MtObject p in pClass.InstancesEnumerator<MtObject>())
{
    p.DeepRemove();
}
```

Removing Index

This example shows how to remove an index from the database schema. The `DeepRemove()` method defined on `MtIndex` will delete the Index. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Removing 'CountryIdx'...");

// Retrieve the index
MtIndex cIdx = MtIndex.Get(db, "CountryIdx");

// Remove the index
cIdx.DeepRemove();

db.Commit();
```

Removing Attributes

This example shows how to remove an attribute from the database schema class. The `DeepRemove()` method defined on `MtAttribute` will delete the attribute and its properties and entry point dictionaries. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Removing 'Country' attribute from 'PostalAddress'
class...");

// Retrieve the class
MtClass paClass = MtClass.Get(db, "PostalAddress");

// Retrieve the attribute
MtAttribute cAtt = MtAttribute.Get(db, "Country", paClass);

// Remove the Attribute
cAtt.DeepRemove();

db.Commit();
```

Removing Relationships

This example shows how to remove an attribute from the database schema class. The `DeepRemove()` method defined on `MtRelationship` will delete the relationship and its inverse relationship. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();

Console.WriteLine("Removing 'Friends' relationship from 'Person'
class...\n");

// Retrieve the class
MtClass pClass = MtClass.Get(db, "Person");

// Retrieve the relationship
MtRelationship fRshp = MtRelationship.Get(db, "Friends", pClass);

// Remove the relationship and its inverse (itself)
fRshp.DeepRemove();

Console.WriteLine("Removing 'SpecialProjectTeam' relationship from 'Manager'
class...");
Console.WriteLine("and its inverse 'SPTeamMember' relationship from
'Employee' class...\n");

// Retrieve the class
MtClass mClass = MtClass.Get(db, "Manager");

// Retrieve the relationship
MtRelationship sptRshp = MtRelationship.Get(db, "SpecialProjectTeam",
mClass);

// Remove the relationship and its inverse
sptRshp.DeepRemove();

Console.WriteLine("Removing 'DeputyManager' relationship from 'Manager'
class...");
Console.WriteLine("and its inverse 'DeputyManagerOf' relationship from
'Employee' class...\n");

MtRelationship dmRshp = MtRelationship.Get(db, "DeputyManager", mClass);

// Remove the relationship and its inverse
dmRshp.DeepRemove();

db.Commit();
```

Removing Classes

This example shows how to remove a class from the database schema. The `DeepRemove()` method defined on `MtClass` will delete the class and its properties and indexes. The connection needs to be open in `MtDatabase.DATA_DEFINITION` mode.

```
// open connection in DDL mode
db.SetOption(MtDatabase.ConnectionOption.DATA_ACCESS_MODE,
(int)MtDatabase.DataAccessMode.DATA_DEFINITION);
db.Open();

db.StartTransaction();
MtClass paClass = MtClass.Get(db, "PostalAddress");
if (paClass != null)
{
    Console.WriteLine("Removing " + paClass.MtClass.MtName +
                              " " + paClass.MtName +
                              " (#" + paClass.MtOid + ")...");

    paClass.DeepRemove();
}
db.Commit();
db.Close();
```

10 Working with Database Events

This section illustrates Matisse Event Notification mechanism. The sample application is divided in two sections. The first section is event selection and notification. The second section is event registration and event handling.

Running EventsExample

This example creates several events, then manipulates them to illustrate the Event Notification mechanism. To run the sample program, follow the instructions below.

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 `Events` directory.
3. Open `Reflection.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `EventNotification` application.
4. To run the example, you need to open at least 2 command-line windows. In each command-line window, change to the appropriate `EventNotification bin` directory and run the applications:

```
> EventNotification localhost example N

> EventNotification localhost example S
```

Events Subscription

This section illustrates event registration and event handling. Matisse provides the `MtEvent` class to manage database events. You can subscribe up to 32 events (`MtEvent.Event.EVENT1 to MtEvent.Event.EVENT32)` and then wait for the events to be triggered.

```
public const uint TEMPERATURE_CHANGES_EVT = (uint)MtEvent.Event.EVENT1;
public const uint RAINFALL_CHANGES_EVT = (uint)MtEvent.Event.EVENT2;
public const uint HIMIDITY_CHANGES_EVT = (uint)MtEvent.Event.EVENT3;
public const uint WINDSPEED_CHANGES_EVT = (uint)MtEvent.Event.EVENT4;

MtDatabase dbcon = new MtDatabase(hostname, dbname);

// Open the connection to the database
dbcon.Open();

MtEvent subscriber = new MtEvent(dbcon);

// Subscribe to all 4 events
ulong eventSet = TEMPERATURE_CHANGES_EVT |
                 RAINFALL_CHANGES_EVT |
                 HIMIDITY_CHANGES_EVT |
                 WINDSPEED_CHANGES_EVT;
```

```
subscriber.Subscribe(eventSet);

ulong triggeredEvents;
int i = 1;
while (i < 20)
{
    // Wait 1000 ms for events to be triggered
    // return false if not event is triggered until the timeout is reached
    if (subscriber.Wait(1000, out triggeredEvents))
    {
        Console.WriteLine("Events (#" + i + ") triggered:");
        Console.WriteLine((((triggeredEvents & TEMPERATURE_CHANGES_EVT) > 0)
? "" : "No ") +
                          "Change in temperature");
        Console.WriteLine((((triggeredEvents & RAINFALL_CHANGES_EVT) > 0) ?
"" : "No ") +
                          "Change in rain fall");
        Console.WriteLine((((triggeredEvents & HIMIDITY_CHANGES_EVT) > 0) ?
"" : "No ") +
                          "Change in humidity");
        Console.WriteLine((((triggeredEvents & WINDSPEED_CHANGES_EVT) > 0) ?
"" : "No ") +
                          "Change in wind speed\n");
    }
    else
    {
        Console.WriteLine("No Event received after 1 sec\n");
    }
    i++;
}

Console.WriteLine("Unsubscribe to 4 Events");
// Unsubscribe to all 4 events
subscriber.Unsubscribe();

// Close the database connection
dbcon.Close();
```

Events Notification

This section illustrates event selection and notification.

```
public const uint TEMPERATURE_CHANGES_EVT = (uint)MtEvent.Event.EVENT1;
public const uint RAINFALL_CHANGES_EVT = (uint)MtEvent.Event.EVENT2;
public const uint HIMIDITY_CHANGES_EVT = (uint)MtEvent.Event.EVENT3;
public const uint WINDSPEED_CHANGES_EVT = (uint)MtEvent.Event.EVENT4;

MtDatabase dbcon = new MtDatabase(hostname, dbname);

// Open the connection to the database
```

```
dbcon.Open();

MtEvent notifier = new MtEvent(dbcon);

ulong eventSet;

eventSet = 0;
eventSet |= TEMPERATURE_CHANGES_EVT;
eventSet |= RAINFALL_CHANGES_EVT;

notifier.Notify(eventSet);

// Close the database connection
dbcon.Close();
```

More about MtEvent

As illustrated by the previous sections, the `MtEvent` class provides all the methods for managing database events. The reference documentation for the `MtEvent` class is included in the Matisse .NET Binding API documentation.

11 Working with a Connection Pool

This section illustrates the implementation and usage of a database connection pool. A database connection pool is a straight forward solution to improve the overall performance of a multi-tier database driven application and to control the database resources allocated to the application.

Running MtDatabasePoolManagerExample

This example creates a MtDatabase pool manager, then use it a multi-threaded application to illustrate Matisse Connection pooling mechanism. To run the sample program, follow the instructions below.

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 `Events` directory.
3. Open `Pooling.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `PoolingExample` application.
4. In a command-line window, change to the appropriate `ReflectionExample bin` directory and run the application:

```
> PoolingExample localhost example
```

Implementing a Connection Pool Manager

The `MtDatabaseConnectionPoolManager` class manages a pool of connections to one Matisse database. It controls the maximum number of open connections included in the pool as well as the maximum time in seconds to wait for a free connection. The class provides basically two public methods, one to get a connection from the pool, the second to return it. The entire source code of the class is included in the *Appendix D: Connection Pooling Source Code* section.

```
public class MtDatabasePoolManager {

  public MtDatabasePoolManager (String hostname, String database,
                                String username, String password,
                                int maxConnections, int timeout) {
  }

  public MtDatabase GetConnection() {
  }


  public void RecycleConnection (MtDatabase conn) {
  }

}
```

Get a Connection from the Pool

```
MtDatabasePoolManager poolMgr = new MtDatabasePoolManager(hostname, dbname,
                                                          maxConnections,
                                                          defaultTimeout);
// Get a connection from the Pool
MtDatabase conn = poolMgr.GetConnection();
```

Return a Connection to the Pool

```
// Release the connection back to the Pool
poolMgr.RecycleConnection(conn);
```

12 Working With Object Factories

Using MtPackageObjectFactory

When your persistent classes are defined in a specific namespace 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 while the schema classes are defined in the`Examples.Csharp.Chap_3` 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","Examples.Csharp.Chap_3"));
```

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

<lang namespace>, <assembly name>

`<lang 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 `<lang namespace>`. For example, if the persistent classes are in the namespace `MyComp.Project1` in the assembly `proj1.dll` and the schema classes are defined in `com.project1`, then the constructor would look like

```
new MtPackageObjectFactory("MyComp.Project1,proj1", "com.project1")
```

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

```
Hashtable namespaceMap = new Hashtable();
namespaceMap.Add("MyComp.Project1.App11,proj1", "project1.appl1");
namespaceMap.Add("MyComp.Project1.App12,proj1", "project1.appl2");
namespaceMap.Add("MyComp.Project2,proj2", "project2.appl1");

new MtPackageObjectFactory(namespaceMap)
```

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());
```

13 Working with Data Classes

The Matisse .NET binding provides the capability to generate Data Classes. Data Classes provide a mechanism to copy back and forth property values into application classes that are independent from the Matisse persistence layer. Data classes share with the stub classes all the properties defined in the database schema classes, but do not include any database access methods. Below are a few examples where this feature can be used:

1. Convert objects using serialization into a format that is transmittable over the network in order for example to transfer the data to a remote application as a Web service.
2. Implement the business logic in an application layer independent from the persistence layer.
3. Add an object caching layer independent from persistence layer.

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 and stub classes share the same name. For example assuming your database contains these three classes (`Person`, `Employee`, and `Manager`), the command

```
> mt_dnom -d example stubgen -lang C# -ln ABC.Project1.Schema -adc
ABC.Project1.DataClasses
```

generates three stub class files `Person.cs`, `Employee.cs`, and `Manager.cs` in containing the stub classes, as well as the `ExampleDataClasses.cs` file containing the Data Classes. The generated stub classes include additional methods to copy the values back an forth between the stub class and its Data Class counterpart.

CAUTION: Any changes to the generated Data Class file will be lost if the code is regenerated. The implementation of Data Class extension must be written a separate code file. This is made possible since Data Classes relies on the Partial class and Partial method definitions features of the programming language.

NOTE: When you need to update the database schema, you first upload the schema into the database using 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.

Manipulating Data Classes

Stub Class Methods

There are a few methods used to copy values back and forth between the Data Class layer and the persistent layer. These methods are all defined on stub classes.

```
public virtual T GetDataClassObject<T>(int navigationDepth) where T : class;
```

This method returns a Data Class object of the appropriate type matching the database object. It also copies the attributes values and load the connected objects in the relationships up to the navigation depth level.

```
public virtual void StoreDataClassObject(object dataObj, int
navigationDepth)
```

This method updates the attributes values in the database object and updates the connected objects up to the navigation depth level.

```
public static StubClassName CreateObjectFromDataClassObject(MtDatabase db,
DataClassName dataObj, int navigationDepth)
```

This method creates a database object from Data Class object, and return the database object. It also sets the attributes values and create or updates the connected objects up to the navigation depth level.

```
protected override void LoadDataClassObjectRelationships(object obj, int
navigationDepth)
```

This method defined on `MtObject` is by default empty. The developer is responsible for implementing this method and therefore defining which relationship is loaded through this object.

```
protected override void StoreDataClassObjectRelationships(object dataObj,
int navigationDepth)
```

This method defined on `MtObject` is by default empty. The developer is responsible for implementing this method and therefore defining which relationship is updated through this object.

```
public virtual object CreateDataClassObject()
```

Create a Data Class object from its Database Object counterpart. This method is redefined in each subclass.

```
protected virtual object LoadDataClassObject(int navigationDepth)
```

Create a Data Class object from its Database Object counterpart and load its attribute values and its relationship values up to a certain depth. This method is redefined in each subclass.

```
protected virtual void LoadDataClassObjectAttributes(object dataObj)
```

Copy all the attribute values of this object to a Data Class object. This method is redefined in each subclass.

```
protected virtual void StoreDataClassObjectAttributes(object dataObj)
```

Copy all the Data Class object attribute values of this Database object. This method is redefined in subclasses.

NOTE: When you load or store 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 `LoadDataClassObjectRelationships` method so it copies all the sub-nodes of each node, then calling

```
aRootNode.GetDataClassObject<DataClassNode>(10);
```

will copy the tree up the third level of depth.
(2) You may fall into an infinite loop. Suppose that you have written the `LoadDataClassObjectRelationships` method for Person so it copies the spouse relationship. If your code is simply like this:

```
dataObj.Spouse =
this.Spouse.GetDataClassObject<DataClasses.Person>(navi
Depth);
```

the execution will fall into an infinite loop unless you control the navigation depth.

```
if (naviDepth > 0)
  dataObj.Spouse =
this.Spouse.GetDataClassObject<DataClasses.Person>(navi
Depth-1);
```

Data Class Properties and Methods

The generated properties and methods for Data Classes are listed in the table below.

Category	Name	Description
Constructor	*Class*	Default Constructors
Properties	*Attribute*	Gets and sets attribute value
	Relationship	Gets and sets successor objects for a relationship
	SchemaClassName	Get the Class Name of the corresponding class in the database schema
	MtOid	Get the OID of the associated database object
	RulesOff	To enable or disable rules defined in the Partial Methods
Methods	SetReference	Update the oid field with the newly created database object ID
	On*Attribute*Changing On*Relationship*Changing	Partial Method executed prior to changing a value
	On*Attribute*Changed On*Relationship*Changed	Partial Method executed after the value change has been completed

Data Class objects must be compared using the `MtOid` property. You can't compare Data Class object with the == operator.

```
DataClasses.Person p1;
DataClasses.Person p2;
...
if (p1.MtOid == p2.MtOid)
  System.out.println("Same objects");
```

Data Class `RulesOff` property enables the control of the execution of the application business logic. The `LoadDataClassObject` method defined on the stub class sets `RulesOff` to `true` during the attributes and relationships loading phase to provide a mean to disable the business logic controls.

```
protected override object LoadDataClassObject(int navigationDepth) {
  ABC.Project1.DataClasses.Person obj =
this.Database.DataClassObjectFactory.GetObjectInstance<ABC.Project1.DataCla
sses.Person>(this);
  obj.RulesOff = true;
  LoadDataClassObjectAttributes(obj);
  LoadDataClassObjectRelationships(obj, navigationDepth);
  obj.RulesOff = false;
  return obj;
}
```

Extending the Generated Data Classes

The generated Data Classes relies on the Partial class and Partial method definitions features of the programming language to define your custom business logic. The partial classes define Data Classes which contain partial methods. These methods are the extensibility points that you can use to apply your business logic before and after any property changes. Partial methods can be thought of as compile-time events. The code-generator defines a method signature and calls the methods in the get and set property accessors. However, if you do not implement a particular partial method, then all the references to it and the definition are removed at compile time.

In the implementing definition that you write in your separate code file, you can perform whatever custom logic is required.

For example assuming your database contains the `Employee` class with an `Salary` property, the source code for the Data Class is as follows:

```
public partial class Employee : Person {

    private int _empId;
    private decimal _salary;
...
    #region Extensibility Method Definitions
    partial void OnEmpIdChanging(System.Int32 val);
    partial void OnEmpIdChanged();
    partial void OnSalaryChanging(System.Decimal val);
    partial void OnSalaryChanged();
    #endregion
...
    public decimal Salary {
        get {
            return this._salary;
        }
        set {
            if ((this._salary != value)) {
                this.OnSalaryChanging(value);
                this._salary = value;
                this.OnSalaryChanged();
```

```
            }
        }
    }
...
}
```

In a separate code file, you can implement whatever custom logic is required. For example:

```
public partial class Employee : Person {

    //
    // Business logic on Salary policy
    //
    // 1 - on or after the planed review date
    // 2 - maximum of 10% increase
    //
    partial void OnSalaryChanging(System.Decimal val)
    {
    ...
    }

    //
    // Set the Next Review Date
    //
    partial void OnSalaryChanged()
    {
    ...
    }
}
```

Running DataClassesExample

`DataClassesExample` creates s set of objects into the database, then manipulates them as Data Class objects. To run the sample program, follow the instructions below.

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 `DataClasses` directory and load into the database the database schema `dataclasses.odl`, which is an ODL (Object Definition Language) file.

   ```
   > mt_sdl -d example import --odl -f dataclasses.odl
   ```

3. Open `DataClasses.sln` in Visual Studio .NET and select Build / Build Solution. This will build the stub classes and Data Classes and compile the `DataClassesExample` application.
4. In a command-line window, change to the `DataClassesExample bin` directory and run the application:

   ```
   > DataClassesExample localhost example
   ```

Loading Data Class Objects

This example shows how to load a Data Class from an object in the database.

```
ABC.Project1.DataClasses.Person dataClassPerson;

MtDatabase db = new MtDatabase(hostname, dbname, new
MtPackageObjectFactory("ABC.Project1.Schema,Schema"));

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

// Load a Data Class Object
// NOTE: load connected objects up to one level depth baed upon the
implementation of LoadDataClassObjectRelationships if any
dataClassPerson = x.GetDataClassObject<ABC.Project1.DataClasses.Person>(1);

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

// Now outside of the Persistent layer, you can manipulate Data Class objects
ABC.Project1.DataClasses.Person x = dataClassPerson;

Console.WriteLine("\t" + x.MtOid + " - " + x.FirstName + " " + x.LastName +
                  " from " + (x.Address != null ? x.Address.City : "???") +
                  " is a " + x.GetType().Name);
```

Copying Attributes Values

When a Data Class object is loaded from a persistent object, the `LoadDataClassObjectAttributes` method is called. This method generated in each stub class copies all the Matisse attribute values from the persistent object to the data object. The `RulesOff` property is set to `true` during phase.

Copying Objects in Relationships

When a Data Class object is loaded from a persistent object, the `LoadDataClassObjectRelationships` method is called. By Default, the method is not redefined on the stub class and the virtual method defined on `MtObject` is called. The developer is responsible for implementing this method and therefore defining which relationship is loaded into the Data Class object. The `RulesOff` property is set to `true` during phase.

For example to load the `Address` object when loading a `Person` object, you need to implement the `LoadDataClassObjectRelationships` method on the `Person` stub class as follows:

```
protected override void LoadDataClassObjectRelationships(object obj, int
navigationDepth)
{
    // Base classes may also define the method.
    base.LoadDataClassObjectRelationships(obj, navigationDepth);

    if (navigationDepth > 0)
    {
```

```
        // Cast the object to the specific data class.
        ABC.Project1.DataClasses.Person dataObj =
                (ABC.Project1.DataClasses.Person)obj;
        // Create a new DataClass object for the Address
        dataObj.Address =
this.Address.GetDataClassObject<ABC.Project1.DataClasses.PostalAddress>(nav
igationDepth-1);
    }
}
```

Saving Data Class Object Changes

This example shows how to report Data Class object changes into the database.

```
ABC.Project1.DataClasses.Employee dClsEmp;
...
// Update the Employee's age
dClsEmp.Age = 34;

// In the Persistent layer
db.Open();
db.StartTransaction();

// Retrieve the database object from its OID
ABC.Project1.Schema.Employee updEmp =
    MtObject.GetObject<ABC.Project1.Schema.Employee>(db, dClsEmp.MtOid);

// Update the corresponding Database Object
// NOTE: the last parameter is set to 0, so none of the connected objects are
updated
updEmp.StoreDataClassObject(dClsEmp, 0);

db.Commit();
db.Close();
```

Saving Attribute Value Changes

When a Data Class object is saved into a persistent object, the `StoreDataClassObjectAttributes` method is called. This method generated in each stub class copies all the Matisse attribute values from the Data Class object to the persistent object.

Copying Objects in Relationships

When a Data Class object is loaded from a persistent object, the `StoreDataClassObjectRelationships` method is called. By Default, the method is not redefined on the stub class and the virtual method defined on `MtObject` is called. The developer is responsible for implementing this method and therefore defining which relationship is updated.

For example to store the `Address` object when saving a `Person` object, you need to implement the `StoreDataClassObjectRelationships` method as follows:

```
protected override void StoreDataClassObjectRelationships(object obj, int
navigationDepth)
{
    // Base classes may also define the method.
    base.StoreDataClassObjectRelationships(obj, navigationDepth);

    if (navigationDepth > 0)
    {
        // Cast the object to the specific data class.
        ABC.Project1.DataClasses.Person dataObj =
(ABC.Project1.DataClasses.Person)obj;

        if (dataObj.Address != null)
        {
            if (this.Address != null && this.Address.MtOid ==
dataObj.Address.MtOid)
            {
                // Update the PostalAddress sub-part object
                this.Address.StoreDataClassObject(dataObj.Address, 0);
            }
            else
            {
                // Create the PostalAddress sub-part object
                this.Address =
PostalAddress.CreateObjectFromDataClassObject(this.Database,
dataObj.Address, 0);
            }
        }
    }
}
```

Creating Data Class Objects

This example shows how to create a Data Class object and then create a counterpart object in the database.

```
// Create a new Data Class Manager object and set its values
ABC.Project1.DataClasses.Manager dClsEmp = new
ABC.Project1.DataClasses.Manager();
dClsEmp.SetValues(1234567, 1234, "John", "Doe", "Paris", "75015");

// In the Persistent layer
db.Open();
db.StartTransaction();

ABC.Project1.Schema.Manager mgr =
    ABC.Project1.Schema.Manager.CreateObjectFromDataClassObject(db,
            dClsEmp, 1);

db.Commit();
db.Close();
```

The `CreateObjectFromDataClassObject` method creates a database object from Data Class object, and return the database object. It also sets the attributes values and create or updates the connected objects up to the navigation depth level. It also set the MtOid property of the Data Class object with the oid of the newly created database object.

Adding Logic to Data Class Objects

The generated Data Classes relies on the Partial class and Partial method definitions features of the programming language to define your custom business logic. The partial classes define Data Classes which contain partial methods. Your application business logic can be implemented in the Partial methods defined in a Data Class.

Suppose you want to implement the following rules when raising the salary of an employee:

1. On or after the planed review date
2. Maximum of 10% increase
3. After a salary change the review date is incremented by 1 year

In a separate code file, you can implement the `OnSalaryChanging` and `OnSalaryChanged` methods as follows:

```
partial void OnSalaryChanging(System.Decimal val)
{
    decimal maxPercentRaise = new decimal(1.10);

    _allowedSalaryChange = false;

    if (!this.RulesOff)
    {
        if (val > _salary)
        {
            if (_reviewDate > DateTime.Today)
            {
                string ErrorMessage = "Invalid update - salary change only
after " + _reviewDate;
                throw new System.Exception(ErrorMessage);
            }
            if (val > Decimal.Multiply(_salary, maxPercentRaise))
            {
                string ErrorMessage = "Invalid update - salary change maximum
" + maxPercentRaise + "X";
                throw new System.Exception(ErrorMessage);
            }
        }
        _allowedSalaryChange = true;
    }
}

partial void OnSalaryChanged()
{
```

```
    if (!this.RulesOff)
    {
        if (_allowedSalaryChange)
        {
            DateTime next = DateTime.Today + new TimeSpan(365, 0, 0, 0);
            // Set the next review Date one year from now
            _reviewDate = next;
        }
    }
}
```

NOTE: Data Class `RulesOff` property which enables the control of the execution of the application business logic is checked to avoid executing the rules during the attributes and relationships loading phase.

The execution of the code below raises an exception and the salary is not changed.

```
ABC.Project1.DataClasses.Employee dClsEmp;
...
// Increase the employee salary by 15%
decimal salaryChange = new decimal(0.15);
dClsEmp.Salary = dClsEmp.Salary + Decimal.Multiply(dClsEmp.Salary,
salaryChange);
```

The execution of the following code raises the salary by 7% and the review date is increased by a year.

```
ABC.Project1.DataClasses.Employee dClsEmp;
...
// Increase the employee salary by 7%
decimal salaryChange = new decimal(0.07);
dClsEmp.Salary = dClsEmp.Salary + Decimal.Multiply(dClsEmp.Salary,
salaryChange);
```

Caching Data Class Objects

The manipulation of Data Class objects by the stub class relies on a Data Class Object Factory attached to a connection object. By Default a `MtDatabase` connection contains a `MtCoreDataClassFactory` object. This class implement a Data Class objects cache which guarantee to return a unique Data Class object associated with a database object. This uniqueness is important to preserve the Data Class objects integrity when manipulating relationships with Data Class objects. By Default, the caching mechanism is turned off.

For example loading a Data Class `Manager` object and its connected objects (`Address` and `ReportTo`) may lead to manipulating inconsistent Data Class objects. Let's assume that the Data Class `Manager` object report to himself, and if the caching is turned off, modifying a property in the Data Class `Manager` object is not seen in the Data Class object accessed via the `ReportTo` relationship.

```
ABC.Project1.Schema.Manager mgr;
...
if (!db.DataClassObjectFactory.Caching)
{
```

```
    Console.WriteLine("WARNING Data Class Caching is OFF");
}

// Load a Manager and its direct connection (Address and ReportTo)
ABC.Project1.DataClasses.Manager dClsMgr =
mgr.GetDataClassObject<ABC.Project1.DataClasses.Manager>(1);

// Update the Manager Age
dClsMgr.Age = 25;
//
if (dClsMgr.MtOid == dClsMgr.ReportsTo.MtOid)
{
    if (dClsMgr.Age == dClsMgr.ReportsTo.Age)
    {
        Console.WriteLine("This is expected.");
    }
    else
    {
        Console.WriteLine("This is inconsistent.");
    }
}
```

Now if the Data Class object caching is turned on, running the code above again leads to the expected behavior.

```
// Turn the cache ON
db.DataClassObjectFactory.Caching = true;
// Clear the cache
db.DataClassObjectFactory.ClearDataClassObjectCache();
```

Sharing a Data Class Object Cache

In a multi-threaded application using a pool of connections it may be convenient in some cases to share the Data Class objects cache. To share a Data Class object cache, you just need to create `MtCoreDataClassFactory` object and share it amongst the `MtDatabase` connections in the pool.

```
MtCoreDataClassFactory cache = new MtCoreDataClassFactory();

// Turn the caching ON
cache.Caching = true;

// Share the cache with the connection in the pool.
db1.DataClassObjectFactory = cache;
db2.DataClassObjectFactory = cache;
db3.DataClassObjectFactory = cache;
...
```

Creating a Data Class Object Factory

Implementing the MtDataClassFactoryI interface

The `MtDataClassFactoryI` interface describes the mechanism used by `MtDatabase` to create the most appropriate Data Class object for each Matisse object. This interface also provides the mechanism for caching Data Class Objects.

```
public class MtCoreDataClassFactory : MtDataClassFactoryI
{
    private readonly Dictionary<int, object> dataClassCache = new
Dictionary<int, object>(512);
    private bool _caching = false;

    public bool Caching
    {
        get { return _caching; }
        set { _caching = value; }
    }

    public void ClearDataClassObjectCache()
    {
        dataClassCache.Clear();
    }

    public virtual T GetObjectInstance<T>(MtObject obj) where T : class
    {
        object res = null;

        if (_caching)
        {
            dataClassCache.TryGetValue(obj.MtOid, out res);
            if (res == null)
            {
                res = obj.CreateDataClassObject();
                dataClassCache.Add(obj.MtOid, res);
            }
        }
        else
        {
            res = obj.CreateDataClassObject();
        }

        return (T)res;
    }


    public virtual void AttachInstance<T>(MtObject obj, T dataObj) where T :
class
    {
        object res = null;
```

```
            if (_caching)
            {
                dataClassCache.TryGetValue(obj.MtOid, out res);
                if (res == null)
                {
                    dataClassCache.Add(obj.MtOid, dataObj);
                }
            }
        }

    }
```

14 Creating an Object Factory

Implementing the MtObjectFactoryI interface

The `MtObjectFactoryI` interface describes the mechanism used by `MtDatabase` to create the appropriate .NET object for each Matisse object. Implementing the `MtObjectFactoryI` interface requires to define the `GetObjectInstance()`, the `GetDatabaseClass()` and `GetClassInstance()` methods which respectively return a .NET object based on an oid, return a schema class name mapping a language namespace and return a .NET object based on a class name.

```
class MyAppFactory : MtObjectFactoryI
{
  public virtual T GetObjectInstance<T>(MtDatabase db, int mtOid) where T :
MtObject
  {
    if (mtOid == 0)
      return null;

    (T)return new MtObject(db, mtOid);
  }

  public virtual T GetClassInstance<T>(MtDatabase db, string className) where T
: MtObject
  {
    if (className == null)
      return null;
    MtClass cls = MtClass.Get(db, className);
    if (cls == null)
      return null;

    return (T)new MtObject(cls); // default is MtObject
  }
  private const string NS_REFLECT = "Matisse.Reflect.";

  public string GetDatabaseClass(string netClsName)
  {
      String res = netClsName;
      if (netClsName.StartsWith(NS_REFLECT))
      {
          res = netClsName.Substring(NS_REFLECT.Length);
    }
    return res;
  }

}
```

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 T GetObjectInstance<T>(MtDatabase database, int oid) where
T : MtObject
  {
    if (IsSchemaObject(database, oid))
    {
      return base.GetObjectInstance<T>(database, oid);
    }
    else
    {
      // Create your .NET object as you see fit
      ...
      return (T)anObject;
    }
  }
}
```

15 Working with Versions

For an introduction to Matisse version access and historical versions, refer to *Getting Started with Matisse*.

This example is a very simple demonstration of version access using a simple one-class schema with three attributes:

```
module Examples {

    module Csharp {

        module Version {

            interface Person : persistent
            {
                attribute Integer id;

                attribute String FirstName = "(unset)";
                attribute String LastName = "(unset)";

                mt_index personId
                    criteria {person::Id MT_ASCEND};
            };

        };
    };
};
```

`VersionExample` allows you to create and modify objects. Whenever an object is created or modified, the application automatically creates a new version (database snapshot).

To create an object, use the set command specifying a new `id` value:

```
set id attribute_name value
```

To modify an object, use the set command specifying the `id` value of the object.

```
set id attribute_name value
```

To list all versions, objects, and values, use the dump command:

```
dump
```

Building VersionExample

1. Follow the instructions in *Before Running the Examples* on page 7.
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 `versions.odl` into the database:

```
> mt_sdl -d example import --odl -f versions.odl
```

4. Open `versions.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `VersionsExa`mple application.

Running VersionExample

1. Open a Command Prompt window, change to the appropriate `bin` directory, and enter:

```
> VersionsExample localhost 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:

```
> VersionsExample localhost 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:

```
> VersionsExample localhost 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:

```
> VersionsExample localhost 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:

```
> VersionsExample localhost example set 2 FirstName Jane
```

5. And modify the second object:

```
> VersionsExample localhost 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.

Creating a Version

This example shows how to create a historical version (database snapshot). Versions are created at commit time. You need to provide a version prefix name to the `commit()` method.

```
db.StartTransaction();
Person p = new Person(db);
p.Id = 1;
p.FirstName = "John";
p.LastName = "Doe";
string vname = db.Commit("VerEx");
Console.WriteLine("New Version " + vname + " created");
```

Accessing a Version

This example shows how to access a historical version (database snapshot). You need to provide the version name (or version prefix if unique) to the `startVersionAccess()` method.

```
db.StartVersionAccess("VerEx");
Person p = Person.LookupId(db, id);
Console.WriteLine(p.Id);
Console.WriteLine(p.FirstName);
Console.WriteLine(p.LastName);
db.EndVersionAccess();
```

Listing Versions

This example shows how to list historical versions (database snapshots). The `MtVersionEnumerator` enumerator allows you to enumerate the versions.

```
db.StartVersionAccess();
foreach (string v in db.VersionEnumerator())
{
    Console.WriteLine("Version " + v);
}
db.EndVersionAccess();
```

16 Building an Application from Scratch

This section describes the process for building an application from scratch with the Matisse .NET binding.

Discovering the Matisse .NET Classes

All the .NET binding classes are provided in a single assembly file. The core classes defined in the **Matisse** namespace. These classes manage the database connection, the object factories as well as the objects caching mechanisms. This assembly file also includes the Matisse meta-schema classes defined in the **Matisse.Reflect** namespace as well as the Matisse `ADO.NET` implementation defined in the **Matisse.Sql** namespace. The Matisse .NET API documentation included in the delivery provides a detailed description of all the classes and methods of the binding.

Matisse Client Server

The Matisse .NET binding is comprised of the **`MatisseNet.dll`** assembly that links the binding to the **`matisse.dll`** library that is the core Matisse Client library.

Matisse Lite

Matisse Lite is the embedded version of Matisse DBMS. Matisse Lite is a compact library that implements the server-less version of Matisse. The .NET binding also includes a Lite version of the binding. The Matisse .NET binding is comprised of the **`MatisseLiteNet.dll`** assembly that links the binding to the **`matisselite.dll`** library which implements the server-less version of Matisse.

Matisse LINQ

The Matisse Linq provider is not included in the `MatisseNet` assembly, but it is included in an additional library named **`MatisseNet.Linq.dll`**.

Generating Stub Classes

1. Design a database schema using ODL (Object Definition Language), SQL DDL, 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.
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# -ln MyCompany.MyProject -sn
MyCompany.MyProject
```

If you need to use the Matisse LINQ provider, you must specify the -linq option to generate a Linq Data Context class. By default, the Linq Data Context class name is set to the database name unless you specify its name with the `-linq` command line option. The Linq Data Context class name is generated in a file named `<Linq class>Linq`

```
> mt_dnom -d example stubgen -lang C# -ln TestLinq.Schema -sn TestLinq.Schema
-linq linqExample
```

When you update your database schema later, load the updated schema into the database using your favorite method (ODL or SQL). 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 stub class files will be preserved.

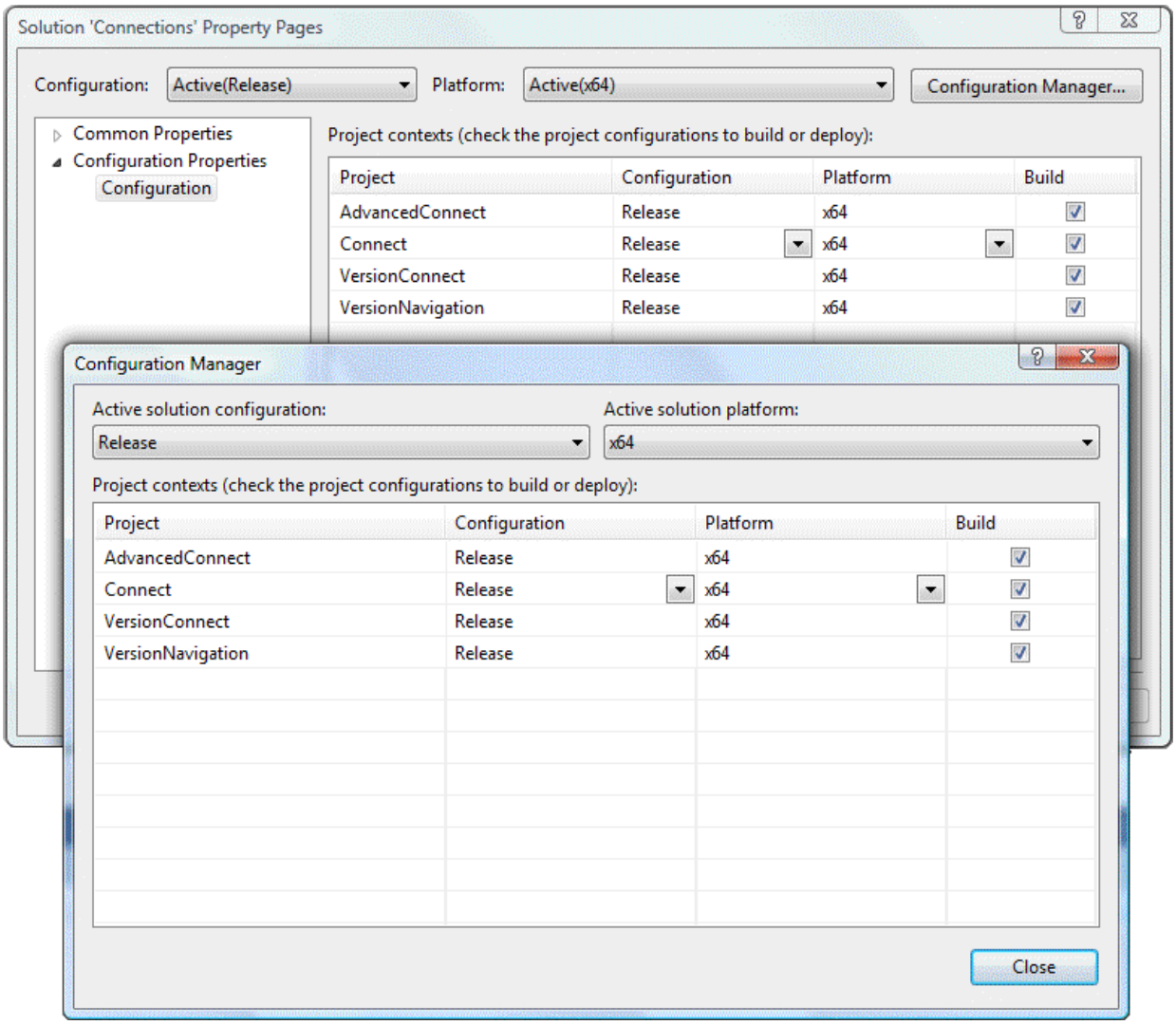
NOTE: The classes generated by the above command are sometimes called "stub classes" to make them distinguished from "data classes", which are explained in *13 Working with Data Classes*.

Creating 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 the Matisse installation directory (i.e. `C:\Program Files\Matisse\bin`), select `MatisseNet.dll`, click Open, and click OK.

   Alternatively, you can reference the library from the command line by entering `csc /lib:<install path>\bin /reference:MatisseNet.dll [/target:library] [/out:somename]` *ApplicationName*, where *ApplicationName* is your .NET program `.cs` or `.vb` source file.
3. If you need to use the Matisse LINQ provider, set the project to reference `MatisseNet.Linq.dll`, the Matisse LINQ library.
4. Add each of the generated class files to the project (right-click on the project icon and select Add / Add Existing Item).
5. Update the Platform settings (`x64, x86`) in the Configuration Properties of your Solution. The Matisse Assemblies comes in 64-bit (`x64`) and 32-bit (`x86`). You need to define the target plaform for your application based upon the Matisse assemblies you have selected for your application.

Define the platform to `x64` for Matisse assemblies 64-bit or define the platform to `x86` for Matisse assemblies 32-bit.



Creating a New Solution for Matisse Lite

The steps for creating a solution for an embedded application using Matisse Lite are identical to the ones described in the section above. You need to substitute **`MatisseNet.dll`** with **`MatisseLiteNet.dll`.** You also need to have the **`matisselite.dll`** library which implements the server-less version of Matisse installed on the machine.

NOTE: The .NET binding API for Matisse Client Server and for Matisse Lite are totally identical making your application working with either one without any code changes.

CAUTION: The delivery of a.NET embedded application using Matisse Lite must include at least both the **`MatisseLiteNet.dll`** and the **`matisselite.dll`** library.

Extending the Generated Stub Classes

You can add your own source code outside of the (`GEN_START GEN_END`) markers produced in the generated stub class. For programming languages supporting regions the generated code is produced in the `'Matisse Generated Code - Do not modify'` region.

```
//  GEN_START: Matisse Generated Code - Do not modify


#region Matisse Generated Code - Do not modify
//  Generated with Matisse .NET Object Manager x64 Version 9.0.0


#endregion

//  GEN_END: Matisse Generated Code - Do not modify
```

When the programming language supports Partial class definitions, the generated stub class takes advantage of this feature. You can add then your own source code to the class without having to modify the generated source file.

For example in C#, the generated stub class for the Person class is as follows:

```
//  Matisse Generated Class
/// <summary>
/// Person is a schema class generated by Matisse.
/// </summary>
//  Matisse Generated Class Inheritance Description - Do not modify
public partial class Person : MtObject {

    //  GEN_START: Matisse Generated Code - Do not modify


    #region Matisse Generated Code - Do not modify
    //  Generated with Matisse .NET Object Manager x64 Version 9.0.0
...
    #endregion

    //  GEN_END: Matisse Generated Code - Do not modify


    //  Generated constructor
    /// <summary>
    /// Default constructor provided as an example.
    /// NOTE: You may modify or delete this constructor
    /// </summary>
    /// <param name="db">a database</param>
    public Person(MtDatabase db) :
```

```
            base(GetClass(db)) {
    }

}
```

You can add your own source code in a different file (i.e. `PersonExtended.cs`) as follows:

```
public partial class Person : MtObject {

    /// <summary>
    /// Cascade Remove method which remove the Address object as well
    /// </summary>
    public override void DeepRemove()
    {
        if (Address != null)
            Address.DeepRemove();
        base.DeepRemove();
    }
}
```

17 Code Generation

As explained in *Generating Stub Classes*, 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.

Code Nomenclature

Stub Class

The generated properties and methods for stub classes are listed in the table below.

Category	Name	Description
Constructor	*Class*	Default Constructor
Properties	*Attribute*	Gets and sets attribute value
	Relationship	Gets and sets successor objects for a relationship
	*Relationship*Size	Gets the number of successor objects for a relationship
Methods	GetInstancesNumber	Returns the number of instances of the class and its all subclasses
	GetOwnInstancesNumber	Returns the number of instances of the class only (excluding its subclasses)
	InstancesEnumerator	Returns an enumerator to iterate through all the instances of the class and its all subclasses
	OwnInstancesEnumerator	Returns an enumerator to iterate through all the instances of the class only (excluding its subclasses)
	*Relationship*Enumerator	Returns an enumerator to iterate through all the successor objects for a relationship
	Prepend*Relationship* Append*Relationship* Remove*Relationship* Clear*Relationship*	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
	Remove*Attribute*	The attribute's value becomes "Unspecified" or falls back to the default value if there is one defined for the attribute
	Is*Attribute*Null	Return true if the value is `MT_NULL` (generated only for Nullable values)
	Is*Attribute*DefaultValue	Return true if the property value is the default value

Category	Name	Description
	Lookup*Index* Lookup*EntryPointDictionary*	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.
	LookupObjects*Index* LookupObjects*EntryPointDictionary*	Searches for objects with the specified key value(s) using the index or the entry-point dictionary. Returns the matching objects.
	Get*Index*ObjectNumber Get*EntryPointDictionary*ObjectNumber	Returns the object count matching the specified key value(s) using the index
	*Index*Enumerator *EntryPointDictionary*Enumerator	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
	CreateDataClassObject	Create a Data Class object of the appropriate type
	LoadDataClassObject	Copy the values of this object to a Data Class object, and return the Data Class object
	LoadDataClassObjectAttributes	Copy the object attribute values into a Data Class object
	StoreDataClassObjectAttributes	Copy the Data Class object attribute values into the object attribute values
	CreateObjectFromDataClassObject	Create a database object from Data Class object, and return the database object

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

Data Class

The generated properties and methods for Data Classes are listed in the table below.

Category	Name	Description
Constructor	*Class*	Default Constructors
Properties	*Attribute*	Gets and sets attribute value
	Relationship	Gets and sets successor objects for a relationship
	SchemaClassName	Get the Class Name of the corresponding class in the database schema
	MtOid	Get the OID of the associated database object
	RulesOff	To enable or disable rules defined in the Partial Methods

Category	Name	Description
Methods	SetReference	Update the oid field with the newly created database object ID
	On*Attribute*Changing On*Relationship*Changing	Partial Method executed prior to changing a value
	On*Attribute*Changed On*Relationship*Changed	Partial Method executed after the value change has been completed

Linq Data Context Class

The generated properties for the Data Context Class are listed in the table below.

Category	Name	Description
Constructor	*DataContextClass(MtDatabase)*	Default Constructors
Properties	*SchemaClass[s,ies,etc.]*	Gets the Entity Table associated with the schema class. The entity table name comes from the pluralized schema class.

Mapping SQL method

SQL methods are mapped to methods in stub class and then can be directly invoked in the .NET application without using a SQL statement. Here is an example of a 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;
```

The stub class for the Manager class contains the following method:

```
public decimal CalculateBonus(double arg1) {
  Decimal res;
  MtCommand dbcmd = ((MtCommand)(this.Database.CreateCommand()));
  dbcmd.CommandText = BuildSQLCallStmt("Manager", "CalculateBonus", 1,
            new Object[] {arg1},
            new MtType.BasicType[] {MtType.BasicType.DOUBLE},
            new MtType.CharEncoding[] {MtType.CharEncoding.CODE_ASCII});
  res = ((Decimal)(dbcmd.ExecuteScalar()));
  dbcmd.Dispose();
  return res;
}
```

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);
```

The mt_dnom Utility

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

```
$ mt_dnom
Matisse .NET Object Manager x64 Version 9.0.0.0 (64-bit Edition) - Feb 17
2012.
(c) Copyright 1992-2012 Matisse Software Inc. All rights reserved.

Usage:
  mt_dnom -d <dbname> [-p] stubgen [-lang C# | VB] [-sn <namespace>] [-ln
<namespace>] [-adc <namesp
ace>] [-linq [<classname>]] [-[no]psm]
    -d    Database [<user>:]<database>[@<host>[:<port>]]
    -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.

    -lang C#  Generate C# files from the database schema.
    -lang VB  Generate VB files from the database schema.
    -sn       Specify the schema class namespace that is mapped to a language
              class namespace.
    -ln       Specify the language class namespace for the generated stub
              classes. when the -sn and -ln options are omitted, each
              class is generated in a namespace matching the schema
              class namespace.
    -adc      Generate ADO Data Classes in addition to stub classes in a
              namespace different from the language class namespace.
    -linq     Generate the LINQ Data Context Class in addition to stub
              classes. If <classname> is omitted, the LINQ Class name is set
              to the database name.
    -[no]psm  Generate .NET methods mapping Persistent SQL methods.
              The default is -psm
```

Stub classes are generated in the directory where the `mt_dnom` utility is executed. Data classes are generated in a single file named `<dbname>DataClasses`.

The Linq Data Context class is generated in the directory where the `mt_dnom` utility is executed. By default, the Linq Data Context class name is set to the database name and is generated in a file named `<Linq class>Linq`.

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.

18 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.

Running StubGen Example

To run the sample program, follow the instructions below.

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 `StubGen` 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 -f examples.odl
   ```

3. Open `StubGen.sln` in Visual Studio .NET and select Build / Build Solution. This will compile the `StubGenExample` application.
4. In a command-line window, change to the appropriate `ReflectionExample bin` directory and run the application:

   ```
   To Generate C# code:

   > StubGenExample example C# -ln MyCompany.MyApp.Schema -sn
   MyCompany.MyApp.Schema
   ```

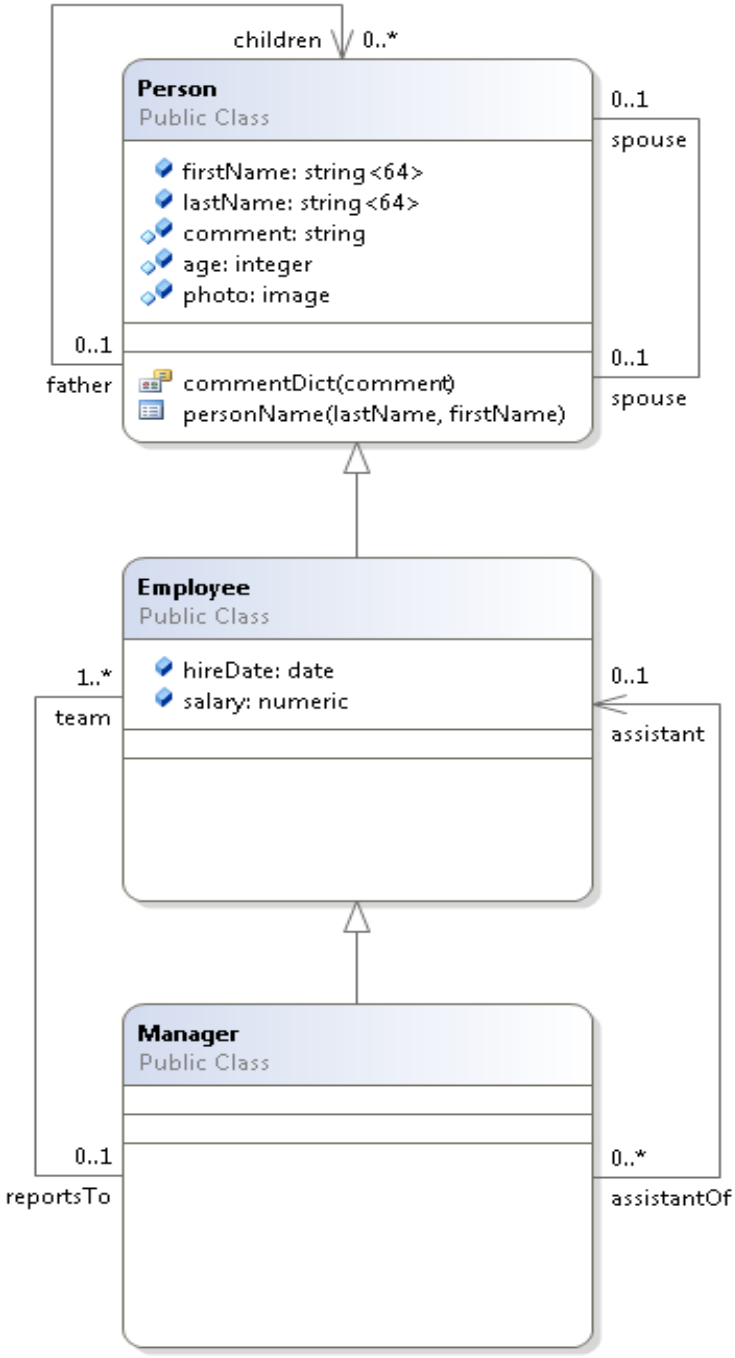
Building a Class Stub Generator

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,
dbNamespace, langNamespace, adcNamespace, nopsm, utilityDesc);
if (string.IsNullOrEmpty(dbNamespace))
{
    foreach (MtClass cls in
MtClass.GetClass(db).InstancesEnumerator<MtClass>())
    {
        if (!cls.IsPredefined())
        {
            engine.GenerateSchemaClassStubFile(cls);
```

```
        }
    }
}
else
{
    // Restricted to this namespace
    MtNamespace nsRoot = MtNamespace.Get(db, dbNamespace);
    if (nsRoot != null)
    {
        GenerateStubs(nsRoot, engine);
    }
    else
    {
        Console.WriteLine("");
        Console.WriteLine("ERROR: database namespace '" + dbNamespace + "' is not
found.");
        Console.WriteLine("");
        return STS_ERROR;
    }
    }

if (!string.IsNullOrEmpty(adcNamespace))
{
    engine.GenerateDataClassesFile(dbname);
}
if (linqOption)
{
    if (string.IsNullOrEmpty(linqClsName))
    {
        linqClsName = dbname;
    }
    engine.GenerateLinqFile(linqClsName, dbname);
}

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

Appendix A: Example.odl Schema




```
module Examples {

    module Csharp {

    module ADO {
```

```
      interface Person : persistent
      {
        attribute String<64> firstName;
        attribute String<64> 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 Set<Person> children inverse Person::father;
        mt_index personName
          criteria {person::lastName MT_ASCEND},
        {person::firstName MT_ASCEND};
        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 Set<Manager> assistantOf inverse
Manager::assistant;
        relationship Manager reportsTo[0,1] inverse Manager::team;
      };

      interface Manager : Employee : persistent
      {
        relationship Set<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:

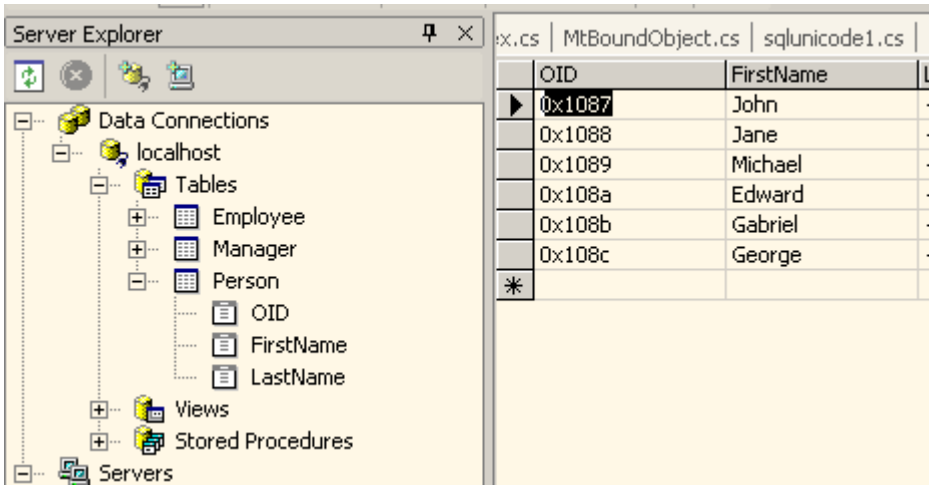
```
MtDatabase dbcon;
dbcon = new MtDatabase (...);

// 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.



Appendix D: Connection Pooling Source Code

```
using System;
using System.Collections.Generic;
using System.Threading;
using Matisse;

namespace PoolingExample
{

    public class RuntimeException : SystemException
    {
        public RuntimeException(string msg, Exception e)
            : base(msg, e)
        {
        }
    }

    public class AssertionError : SystemException
    {
        public AssertionError()
            : base()
        {
        }
    }

    /// <summary>
    /// simple MtDatabase pool manager.
    /// </summary>
    public class MtDatabasePoolManager : IDisposable
    {
        private string hostname;
        private string database;
        private string username;
        private string password;
        private int maxConnections;
        private int timeout;
        private Semaphore cxnAvailable;
        private Stack<MtDatabase> recycledConnections;
        private int activeConnections;
        private bool isDisposed;

        /// <summary>
        /// Constructs a MtDatabasePoolManager object with a timeout of 10
seconds.
        /// </summary>
        /// <param name="hostname">the host name (or IP address).</param>
        /// <param name="database">the database name.</param>
        /// <param name="maxConnections">the maximum number of
connections.</param>
```

```
        public MtDatabasePoolManager(string hostname, string database, int
maxConnections)
            : this(hostname, database, null, null, maxConnections, 10)
        {

        }

        /// <summary>
        /// Constructs a MtDatabasePoolManager object.
        /// </summary>
        /// <param name="hostname">the host name (or IP address).</param>
        /// <param name="database">the database name.</param>
        /// <param name="maxConnections">the maximum number of
connections.</param>
       /// <param name="timeout">the maximum time in seconds to wait for a free
connection..</param>
        public MtDatabasePoolManager(string hostname, string database, int
maxConnections,
                                     int timeout) :
            this(hostname, database, null, null, maxConnections, timeout)
        {
        }

        /// <summary>
        /// Constructs a MtDatabasePoolManager object.
        /// </summary>
        /// <param name="hostname">the host name (or IP address).</param>
        /// <param name="database">the database name.</param>
        /// <param name="username">the user name, or <code>null</code></param>
        /// <param name="password">the user's password, or
<code>null</code>.</param>
        /// <param name="maxConnections">the maximum number of
connections.</param>
       /// <param name="timeout">the maximum time in seconds to wait for a free
connection..</param>
        public MtDatabasePoolManager(string hostname, string database, string
username,
                                 string password, int maxConnections, int timeout)
        {
            this.hostname = hostname;
            this.database = database;
            this.username = username;
            this.password = password;
            this.maxConnections = maxConnections;
            this.timeout = timeout;
            if (maxConnections < 1)
                throw new ArgumentOutOfRangeException("Invalid maxConnections
value.");
            cxnAvailable = new Semaphore(maxConnections, maxConnections);
            recycledConnections = new Stack<MtDatabase>();
        }

        ~MtDatabasePoolManager()
        {
```

```
            Dispose(false);
        }


        /// <summary>
        /// Closes all unused pooled connections.
        /// </summary>
       /// <param name="disposing">a flag to dispose the managed resources of the
class</param>
        protected void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Code to dispose the managed resources of the class
            }
            // Code to dispose the un-managed resources of the class
            lock (this)
            {
                if (isDisposed) return;
                isDisposed = true;
                MtException e = null;

                while (0 != recycledConnections.Count)
                {
                    MtDatabase dbconn = recycledConnections.Pop();
                    try
                    {
                        dbconn.Close();
                    }
                    catch (MtException e2)
                    {
                        if (e == null) e = e2;
                    }
                }
                if (e != null) throw e;
            }

            isDisposed = true;
        }

        /// <summary>
        /// Closes all unused pooled connections.
        /// </summary>
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        /// <summary>
        /// Retrieves a connection from the connection pool.
        /// If <code>maxConnections</code> connections are already in use, the
method
```

```
        /// waits until a connection becomes available or <code>timeout</code>
seconds elapsed.
       /// When the application is finished using the connection, it must close it
        /// in order to return it to the pool.
        /// throws TimeoutException when no connection becomes available within
<code>timeout</code> seconds.
        /// </summary>
       /// <param name="disposing">a flag to dispose the managed resources of the
class</param>
        /// <return>return a new Connection object</return>
        public MtDatabase getConnection()
        {
            lock (this)
            {
                if (isDisposed) throw new InvalidOperationException("Connection
pool has been disposed.");
            }
            try
            {
                if (!cxnAvailable.WaitOne(timeout * 1000))
                    throw new TimeoutException("Timeout while waiting for a free
database connection.");
            }
            catch (ThreadInterruptedException e)
            {
                throw new RuntimeException("Interrupted while waiting for a
database connection.", e);
            }
            bool ok = false;
            try
            {
                MtDatabase conn = getPooledConnection();
                ok = true;
                return conn;
            }
            finally
            {
                if (!ok) cxnAvailable.Release();
            }
        }

        private MtDatabase getPooledConnection()
        {
            lock (this)
            {
                if (isDisposed) throw new InvalidOperationException("Connection
pool has been disposed.");
                MtDatabase conn;
                if (0 != recycledConnections.Count)
                {
                    conn = recycledConnections.Pop();
                }
                else
                {
```

```
                    // The third argument is given so that the connection object
can find
                    // the persistent classes, which are defined in a the
"Mycomp.Myapp" namespace in the "Schema" Assembly.
                    // conn = new MtDatabase(hostname, database, new
MtPackageObjectFactory("Mycomp.Myapp,Schema"));

                    conn = new MtDatabase(hostname, database);
                    conn.Open(username, password);
                }
                activeConnections++;
                assertInnerState();
                return conn;
            }
        }

        /// <summary>
        /// Recycle a connection into the connection pool.
        /// </summary>
        /// <param name="conn">a Connection object</param>
        public void recycleConnection(MtDatabase conn)
        {
            lock (this)
            {

                if (isDisposed)
                {
                    disposeConnection(conn);
                    return;
                }
                if (activeConnections <= 0) throw new AssertionError();
                activeConnections--;
                cxnAvailable.Release();
                recycledConnections.Push(conn);
                assertInnerState();
            }
        }

        private void disposeConnection(MtDatabase conn)
        {
            lock (this)
            {
                if (activeConnections <= 0) throw new AssertionError();
                activeConnections--;
                cxnAvailable.Release();
                closeConnectionNoEx(conn);
                assertInnerState();
            }
        }

        private void closeConnectionNoEx(MtDatabase conn)
        {
            try
            {
```

```
                conn.Close();
            }
            catch (MtException)
            {
            }
        }


        private void assertInnerState()
        {
            if (activeConnections < 0)
                throw new AssertionError();
            if (activeConnections + recycledConnections.Count > maxConnections)
                throw new AssertionError();
        }

        /// <summary>
        /// Returns the number of active (open) connections of this pool.
        /// This is the number of <code>Connection</code> objects that have been
        /// issued by <code>getConnection()</code> for which
<code>Connection.close()</code>
        /// has not yet been called.
        /// </summary>
        /// <return>return the number of active connections.</return>
        public int getActiveConnections()
        {
            lock (this)
            {
                return activeConnections;
            }
        }

        public override string ToString()
        {
            return "MtDatabasePoolManager:" +
                "hostname=" + hostname +
                ";database=" + database +
                (username != null ? ";username=" + username : "") +
                (password != null ? ";password=" + password : "") +
                ";maxConnections=" + maxConnections +
                ";timeout=" + timeout;
        }

    }
}
```

Index

P

R

S

T