Getting Started with Matisse

January 2017

Getting Started with Matisse

PDF generated 7 January 2017

Contents

1 Introduction

1.1 Scope of This Document

This document is intended to help new users learn the basics of Matisse design and programming. It serves as an introduction to the *Matisse SQL Programmer's Guide*, *Matisse .NET Programmer's Guide* (including ADO.NET), the *Matisse C++ Programmer's Guide*, the *Matisse Eiffel Programmer's Guide*, the *Matisse PHP Programmer's Guide*, the *Matisse Python Programmer's Guide*, and the *Matisse Java Programmer's Guide*.

1.2 Before Reading This Document

Throughout this document, we presume that you already know the basics of database design and are familiar with the languages you plan to use to develop Matisse applications.

2 Matisse Quickstart

If you have just downloaded Matisse, you can get rapidly a sense of how it works by following the steps in this section which show you how to create a database, populate it with instances, and query the instances you have just created.

Initialize the 'example' database

*   On MS Windows, start the Enterprise Manager, click on 'Programs->Matisse->Enterprise Manager' from the start menu.

*   On Unix, first verify that you have the right environment. To that effect, you may run the mt_env.sh or mt_env.csh script in your install directory. Then start the Enterprise Manager:

    % sh <INSTALL_PATH>/mt_env.sh
    % mt_emgr &

Figure 2.1   The Database View in Matisse Enterprise Manager
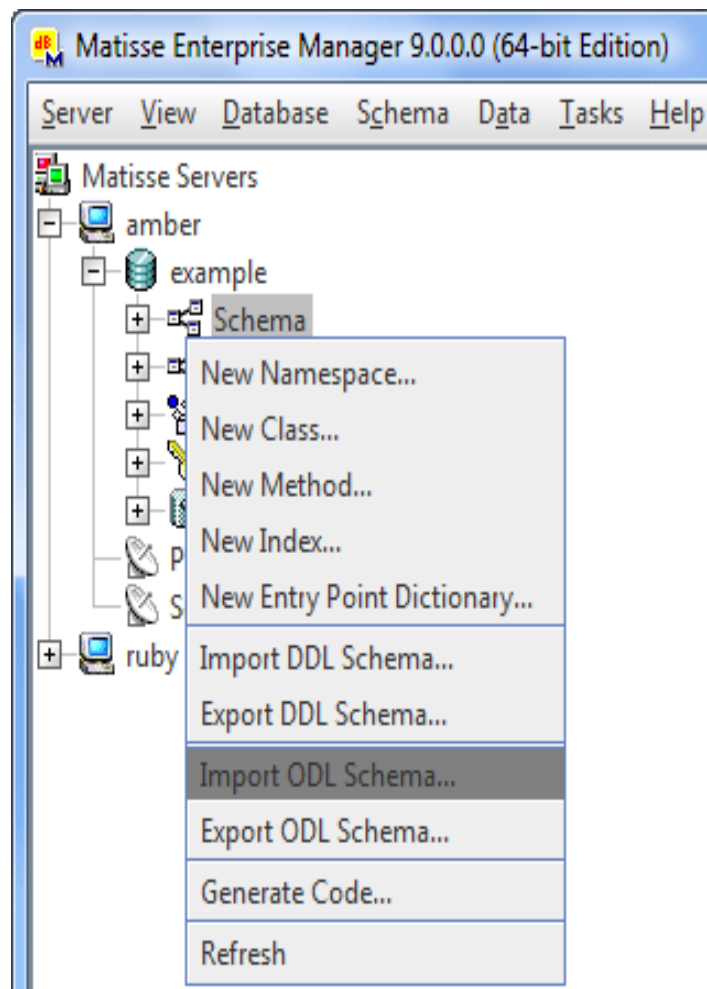
- In the database view, Right click on the 'example' database. Then click on 'Start'. At this point, you have an empty database up and running.

Create instances of the class 'movie'

- Under 'example', open the node 'Data', then click on 'SQL Query Analyzer'. Create a class and insert some instances, you may cut and paste the following statement in the SQL Query Editor window:

CREATE CLASS Movie (Title VARCHAR(64), Rating VARCHAR(4));

Figure 2.2   The SQL Query Analyzer
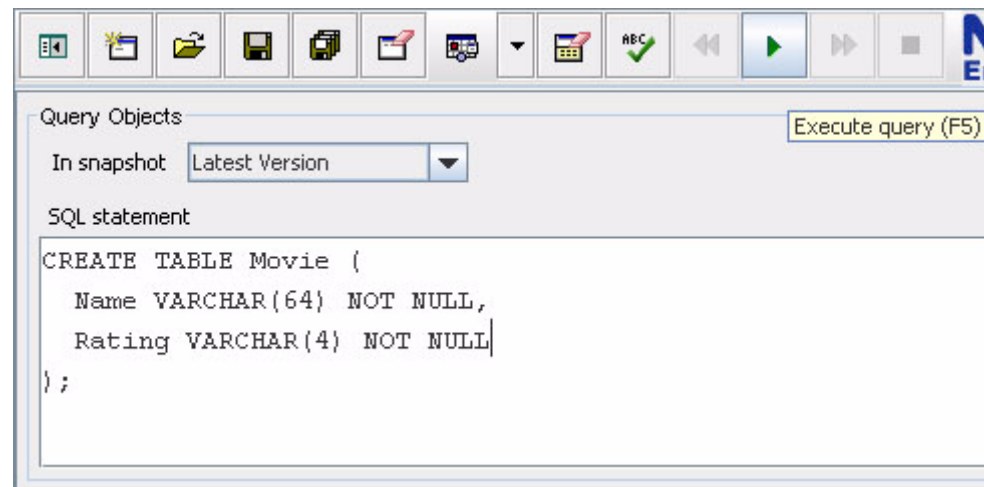
- Click 'Execute Query' and then 'Clear SQL Query Editor' (the left most eraser icon) and enter the following statement:

  INSERT INTO Movie (Title, Rating) VALUES ('Rocky', 'R');

- Click 'Execute Query' and then 'Clear SQL Query Editor' and enter the following statement:

  SELECT * FROM Movie;

- Following the same procedure, enter the following statement to create your first SQL method, which counts the movies according to some rating:

  CREATE STATIC METHOD CountMovies(aRating STRING)
  RETURNS INTEGER
  FOR Movie
  BEGIN
     DECLARE Cnt INTEGER DEFAULT 0;
     SELECT COUNT(*) INTO Cnt FROM Movie WHERE Rating = aRating;
     RETURN Cnt;
  END;

- At this stage, you have created a simple schema in your database 'example' and added some data in the database. In the left side window, you may open the 'Classes' node under 'Schema' to view the schema that you have created.

Figure 2.3   Example Database Schema
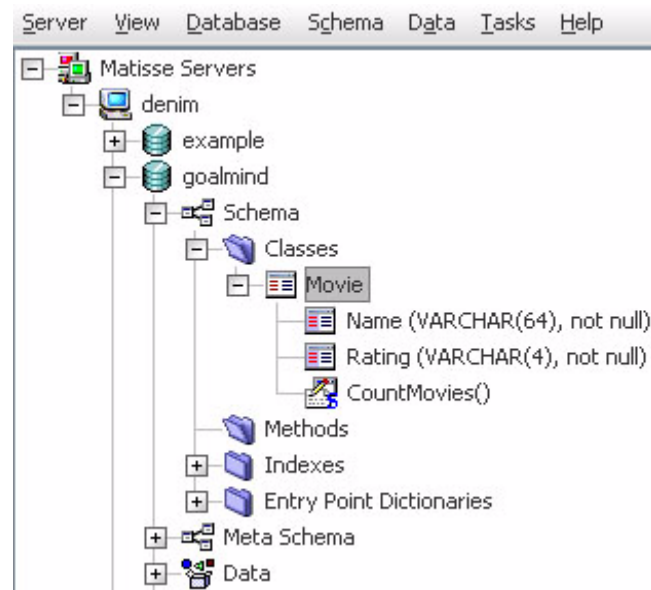
- You can save this schema in DDL (Data Definition Language) format with a right click on the 'Schema' node, then 'Export DDL Schema ...'. It produces a text file that contains your SQL DDL statements.

- You may execute the method 'CountMovies' with a right click on the method node 'CountMovies(STRING)', then substitute the string '<aRating STRING>' with 'R'. The statement should look as follows before you execute it:

  CALL Movie::CountMovies('R');

3   Overview of Matisse Development

A typical Matisse development project will consist of the following steps, not necessarily in this order.

Design

*   Decide which of the various Matisse APIs and bindings (C, C#, C++, Eiffel, Java, Perl, PHP, Python, Smalltalk, Visual Basic, XML, and also ADO.NET, JDBC, or ODBC) are most appropriate for your application.

*   Sketch out the conceptual model for the database to determine what classes, attributes, and relationships will be required.

*   Define the corresponding application schema using SQL Data Definition Language (DDL), the Object Description Language (ODL), or Matisse Modeler tool (UML-like).

*   Write the core logic of your application with SQL Stored Methods. These methods are stored and executed on the server side and can be called from any client environment.

Create a database

*   Create and initialize your database.

*   Load your schema and SQL methods into the database.

Automatic code generation for client applications

*   This step is not necessary if you use the C API or a third party SQL tool. If appropriate, generate C#, C++, Eiffel, Java, or Visual Basic class files corresponding to the schema classes (See the relevant *Matisse Programmer's Guide*).

*   You may produce HTML documentation for the generated C#, C++, or Java classes.

Write the client application code

*   Write the rest of your client application's code, which may include extensions in the user area of the generated class files, and debug.

Maintenance

When your application evolves in such a way that it becomes necessary to change the application schema:

*   With SQL DDL: Use CREATE or ALTER statements to make the changes.

*   With ODL: Revise the ODL definition, then import it from the Enterprise Manager to update the schema in the database.

*   With Matisse Modeler: Revise the model, then export to Matisse.

4 Components of a Matisse Database

4.1 Application Schema

In Matisse, you design a schema for your application using three basic components: classes, attributes, and relationships.

The following UML diagram of a simple database schema illustrates the three components and how they relate:
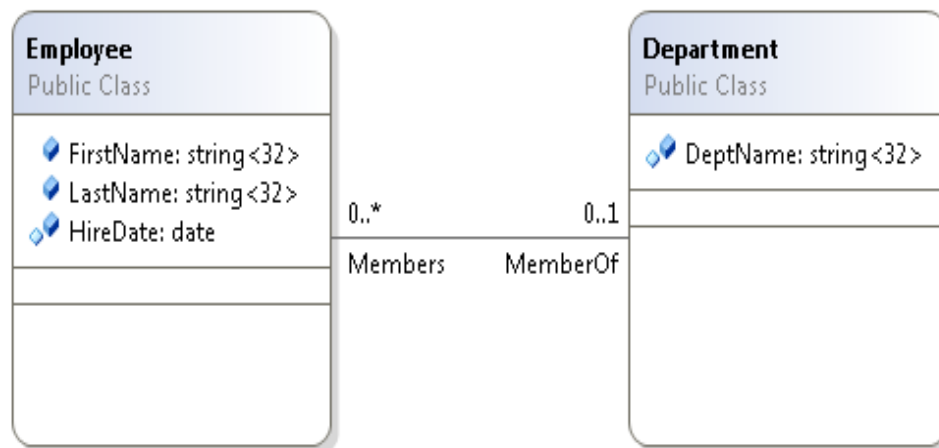
- The classes Employee and Department contain the attributes FirstName, LastName, and HireDate.

- The two schema classes are linked by the relationship Members and its inverse MemberOf. The cardinality indicators (0..* and 0..1) show that this is a one-to-many relationship: a department may have any number (0..*) of members, but an employee may be a member of only one department.

Try it

- You can create this schema with the following statements. Enter the following statements in the 'SQL Query Editor' window:

```
CREATE CLASS Employee (FirstName VARCHAR(32) NOT NULL,
                       LastName VARCHAR(32) NOT NULL,
                       HireDate DATE,
                       MemberOf REFERENCES (Department) CARDINALITY (0, 1)
                               INVERSE Department.Members);
CREATE CLASS Department (DeptName VARCHAR(32),
                       Members REFERENCES SET (Employee) CARDINALITY (0, -1)
                               INVERSE Employee.MemberOf);
```

- Then right click on 'Classes', then 'Refresh', to view the classes you have created. You can create some instances and link them as follows:

```
INSERT INTO Department (DeptName)
        VALUES ('Sales') RETURNING REF(Department) INTO aDepartment;
```

```
INSERT INTO Employee (FirstName, LastName, HireDate, MemberOf)
                VALUES ('John', 'Smith', DATE '2000-10-12', aDepartment);
DROP SELECTION aDepartment;
```

• Then to view the instances that you have created:

```
SELECT p.*, p.MemberOf.DeptName FROM Employee p;
```

<div style="color:blue">relational
database
counterpart:
tables</div>

A schema class is similar to a table in a relational database. For example, the class Employee defines a set of instances, one for each employee, each of which contains attribute values that stores a particular person's employee's name and hire date, just as in a relational database the table EMPLOYEE would contain a set of rows, one for each employee, which would store the same data.

4.2 Class Inheritance

A class may be defined as a subclass that inherits all of its superclasses properties and defines additional attributes and/or relationships.

Multiple inheritance is supported. However, it must be avoided when exporting your classes in C#, Java, or Smalltalk, which do not allow multiple inheritance.

Try it

You can create the Manager class, which inherits from Employee and adds a Team relationship. Enter the following statements in the 'SQL Query Editor' window:

```
CREATE CLASS Manager UNDER Employee (
                Team REFERENCES SET (Employee) CARDINALITY (0, -1)
                        INVERSE Employee.ReportsTo);
ALTER CLASS Employee ADD RELATIONSHIP
                ReportsTo REFERENCES (Manager) CARDINALITY (0, 1)
                        INVERSE Manager.Team;
```

4.3 Referential Integrity and Cardinality Constraints

At right, Employee and Manager have one relationship that defines which employees report to which managers, and another that identifies each manager's assistant.

The cardinality of each end of a relationship is controlled by specifying the minimum and maximum number of successors allowed (with infinity represented by -1 in code and * in diagrams). If a single integer is specified, it is both the minimum and the maximum number. In the example at right:

- An employee must report at most to one manager (0..1).

- A manager's team consists of any number of employees (0..*).

Matisse automatically maintains referential integrity. For example, if an Employee instance is deleted, the reference to the employee is automatically removed from the successors of the Manager instance to which is it associated through the Manager.Team relationship.
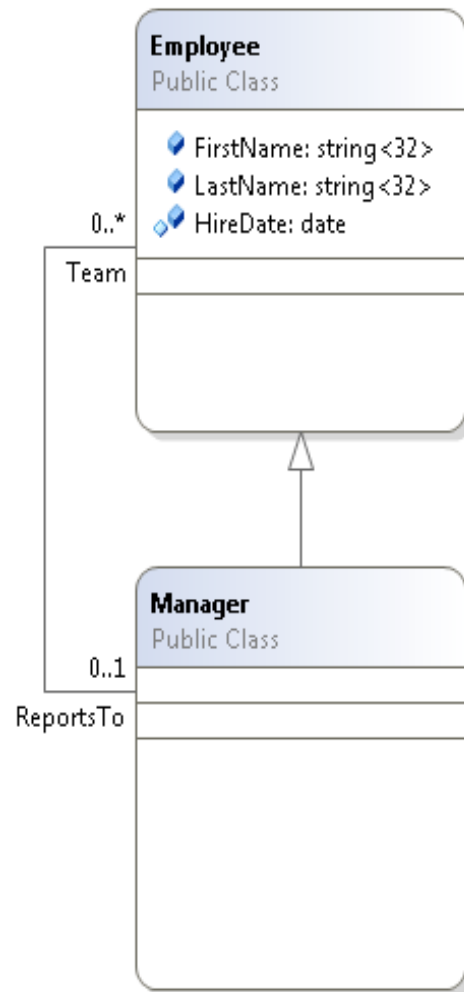
relational database counterpart: foreign keys

Matisse relationships work in a similar way than foreign keys between relational tables, but Matisse relationships make database design simpler and more efficient:

- You do not need to define the additional columns and indexes that are required for making things work with primary keys and foreign keys.

- The definition of many-to-many relationships does not require to use intermediate tables.

- You can use navigational expressions to access data. For example, you could find all the employees who report to a particular manager using the SQL statement "SELECT * FROM Employee WHERE ReportsTo.LastName = 'Steiner'".

4.4 SQL Methods

SQL Stored Methods (or more simply 'SQL methods') are always associated to a class and stored as part of your application schema in the database. Like for other properties of a class, Methods are inherited in subclasses.

You can define either STATIC or INSTANCE methods:

- An INSTANCE method is executed for a given instance and can access or update the instance with the keyword SELF. An instance method that is inherited in a subclass can be overridden as you define another method with the same name and signature in the subclass.

- A STATIC method is not associated at execution time with a particular instance. To call a static method, you must mangle the class name with the method. For instance to call the static method CountMovies on the class Movie: 'CALL Movie::CountMovies('R');'.

SQL Methods are executed on the database server side. Method execution can be invoked interactively with a CALL statement, from within another method, or from within the WHERE clause of a SELECT statement.

Try it

You can create the isRRated instance method on the Movie class:

```
CREATE INSTANCE METHOD isRRated()
RETURNS BOOLEAN
FOR Movie
BEGIN
   IF SELF.Rating = 'R' THEN
      RETURN TRUE;
   ELSE
      RETURN FALSE;
   END IF;
END;
```

Then, you can execute it in a WHERE clause:

```
SELECT m.* FROM Movie m WHERE m.isRRated() = TRUE;
```

4.5 Indexes

Indexes are selected by the Matisse SQL optimizer to speed up the processing for the ORDER BY clause and for the WHERE clause filtering when running equi-match and range queries. You can also use indexes from the C API and the language bindings to lookup instances based on a key value.

For a given class, up to four attributes can be used to form a composite key for an index. The attributes used in an index must be non-nullable. You may add or drop an index at any time, without stopping the database operations.

Matisse indexes are inherited. For instance, an index defined on the class Employee is inherited by the subclass Manager. The following queries will use the same index:

    SELECT * FROM Employee WHERE LastName = 'Steiner';

    SELECT * FROM Manager WHERE LastName = 'Steiner';

    SELECT * FROM ONLY Manager WHERE LastName = 'Steiner';

Try it

The following statement creates an index for the Employee class:

    CREATE INDEX EmpIdx ON Employee (LastName ASC, FirstName ASC);

4.6 Entry-Point Dictionaries

An entry-point dictionary is an indexing structure containing keywords derived from a string value. For instance, you may use an entry-point dictionary to implement full text indexing on an attribute containing some textual data.

- The keywords generated for an entry-point dictionary depend on which make-entry function is selected. Currently two functions are provided: the default make-entry function indexes the full content of the attribute value as a single key, while the full-text-make-entry function generates keys for all the words that are contained in a textual attribute value.

- An entry-point dictionary is automatically updated whenever an update to an associated instance is committed.

- You may add or drop an entry-point dictionary at any time, without stopping the database operations.

For instance, if you have an entry-point dictionary ContentDict defined with the full-text-make-entry function on the attribute Content for the class Book, you can retrieve the titles of all books that contain a particular word:

    SELECT * FROM Book WHERE ENTRY_POINT(ContentDict) LIKE 'Grape%'

Try it

The statements to create the class Book and the associated dictionary ContentDict would be like this:

    CREATE CLASS Book (Title VARCHAR(64), Content TEXT);
    CREATE ENTRY_POINT DICTIONARY ContentDict ON Book (Content)
                MAKE_ENTRY "make-full-text-entry";

4.7 The Meta-Schema

The meta-schema is a pre-defined schema from which the classes and other objects that comprise your application schema are instantiated. For instance:

- schema classes are instantiated from MtClass

- attribute descriptors are instantiated from MtAttribute

- relationship descriptors are instantiated from MtRelationship

One benefit of the meta-schema is that it allows the dynamic discovery of the application schema.

Try it

For instance, to list all the classes, attributes an methods, defined in your database, you may run the queries:

SELECT MtName FROM MtClass;

SELECT MtName FROM MtAttribute;

SELECT MtName FROM MtMethod;

5   Dynamic Schema Evolution

You can update your database schema using SQL DDL, ODL without having to turn off the database, or to migrate the existing data. The only restriction is that the integrity constraints on the existing data must not be violated by adding, removing, or updating property definitions. The next table shows the limitations for each case of schema change when data already exists in the database.

Table 1   Limitations of Schema Update When Data Exists

Type of Change	Description of Limitation
Add/Update an attribute	The new attribute needs to have a default value (a), or it needs to be nullable (b). Examples in SQL DDL: (a) ALTER CLASS Manager ADD ATTRIBUTE Bonus INTEGER **DEFAULT** 0; (b) ALTER CLASS Person ADD ATTRIBUTE Address VARCHAR(250);
Remove an attribute	*no constraint*
Add/Update a relationship	The minimum cardinality of the new relationship needs to be 0. Example in SQL DDL: ALTER CLASS Employee ADD REFERENCES worksIn (Project) CARDINALITY (**0**, 5) INVERSE ...
Remove a relationship	*no constraint*
Add a superclass	*no constraint*
Remove a superclass	*no constraint*

Note that SQL methods in the database may need to be recompiled after a schema change and they need to be consistent with the new schema. For example, if you remove an attribute from a class, make sure that no SQL stored method references the attribute. To recompile all the SQL methods in your database, use the COMPILE ALL statement.

6   Matisse in Operation

This document covers only those aspects of Matisse operation and administration with implications for the programmer. For information on aspects of Matisse operation that are transparent to applications, such as datafile configuration, see the *Matisse Server Administration Guide*.

6.1 Connections

Client applications interact with Matisse through a "connection" that handles basic communication between the client application and the Matisse server application: selecting a server and a particular database, login, starting and committing or aborting transactions, and so on.

*   A Matisse database can handle many simultaneous connections from a variety of clients. For example, a database could simultaneously be accessed from a Web Server with client threads using PHP or JDBC, by standalone clients running a C#, C++, Eiffel, or Java application, and by a SQL ODBC-based reporting tool.

*   A Matisse client can connect to multiple databases by using multiple connections. The Matisse Client library is thread safe and allows you to establish multiple connections in the same thread or in separate threads.

For detailed information about connections, see the relevant API or language binding documentation.

6.2 Transactions

Once a connection with a database has been established, the next step in interacting with Matisse is to open a transaction (read-write access) or a version access (read-only; see *Version Access*). See the relevant API or language binding documentation for examples.

Matisse transactions follow the "ACID" rule:

*   Atomicity: transactions are either committed completely or aborted.

*   Consistency is maintained at all times.

*   Isolation: the results of one transaction are not visible to other transactions until it is committed.

*   Durability: committed transactions are not affected by server or system failure.

When a transaction is committed, Matisse does the following:

*   It checks that all the instances that were created or modified to verify the constraints of the schema: that an attribute of type INTEGER does not have a non-integer or out-of-range value, that a non-nullable attribute with no default value has a value, that a relationship with a maximum cardinality of 1 does not have more than one successor, and so on.

*   If any check fails, Matisse returns an error message identifying the invalid instance. The transaction is not aborted; if the client application knows how to handle the error, it can fix it and try to commit again.

- If all checks pass, the modified instances are written in the database and the transaction is validated.

- If specified by the commit instruction, a new named version of the database is created.

6.3 Database Locks

Matisse automatically sets read and write locks on individual instances. The locking rules are as follows:

- Several transactions can read an attribute or relationship simultaneously.

- A transaction cannot modify an attribute or relationship while it is being read or modified by another transaction. An attempt to make such a modification will be blocked by the server until the other transaction is committed or aborted. If the optional LOCK_WAIT_TIME argument was passed when initiating the connection, the attempt will fail if the time-out expires before the other transaction releases its locks.

- A transaction *can* modify an attribute or relationship that is being read by a version access; (Matisse will preserve an unmodified copy for use by the version access).

When two transactions have read a particular instance property and then both transactions try to modify it, a deadlock arises. For example:

1. Client 1 reads a value, establishing a read lock on the attribute.

2. Client 2 reads the same value, establishing a second read lock on the attribute.

3. Client 1 attempts to update the value. Since client 2 has a read lock, the server holds this attempt pending commit or abort of client 2's transaction.

4. Client 2 attempts the same update. Since client 1 has a read lock, the server holds this attempt as well, pending commit or abort of client 1's transaction.

Thus each transaction is being held pending commit or abort of the other. Matisse resolves such deadlocks by aborting:

- the transaction with the lower priority (a priority from 1 to 8 may optionally be specified when opening a transaction);

- or, if the priorities are the same, the transaction that has performed the fewest updates;

- or, if the transactions have performed the same number of updates and have the same priority, the transaction that caused the deadlock (that is, that attempted the update later).

Such deadlocks can be prevented by either setting an explicit write lock on a given instance, or by using the 'Read For Update' transaction mode. In this mode, all read accesses set write locks in order to be able to update without upgrading from shared mode to exclusive mode.

All locks are released when the transaction is either committed or aborted.

> **NOTE:**   You can take advantage of Matisse version access for data intensive tasks like data analysis that would require locking or dirty reads in other database products.

6.4 Version Access

When a version access (read-only transaction) is opened, all the data stored in the database at that moment is preserved until the access is closed. This requires no locks or wholesale copying: as instances are modified by transactions, Matisse preserves unmodified copies for use by the version access.

The figure at right represents a new database in which a transaction (T1) has created three instances, A, B, and C. If a version access is opened on this database, then a second transaction (T2) modifies instance B, deletes instance C, creates a new instance D, and commits. The figure below shows the result:

Ⓐ Ⓑ Ⓒ

▨ modified since version access started
▬ unmodified

what a current transaction sees

Ⓐ Ⓑ Ⓓ

what the version access sees

Ⓐ Ⓑ Ⓒ

In other words, a coherent "snapshot" of the database at the time each version access was started is preserved without interfering with current transactions' ability to modify instances. Once the version access has closed, the preserved copies of old instances will be eligible for garbage collection.

For more about versions, see the *Matisse Server Administration Guide*.

6.5 Named Versions

Named versions (sometimes called "savetimes") preserve such "snapshots" of a database indefinitely. Within an application, named versions can be registered when committing a transaction.

•   When declaring a named version, the client application provides a name to identify it.

•   To generate the full name for the version, Matisse appends a unique ID number to the name that is provided. (This ensures that two clients will not create two versions with the same full name.)

•   The version is accessed by passing its full or partial name as an argument when opening a version access.

•   Copies of modified instances preserved by Matisse to maintain the named version are not eligible for garbage collection until the version is manually undeclared using the mt_version command (see the *Matisse Server Administration Guide*).

6.6 Client Cache Management

By default Matisse transparently caches data on the client side to optimize access to the network. We describe here the different caching alternatives that are currently implemented in Matisse.

Accessing instances in a Client Application

When you access one attribute value or another property of an instance, the instance's other properties are also cached. When you access another property for the same instance, the Matisse client retrieves it from the cache.

When a transaction is committed, the updates are flushed from the client cache to the server. When a transaction is committed, aborted or a version access is ended, any value read is deleted from the cache. This ensures that the cache is always consistent with the database.

Streaming Attributes

You may skip the default caching mechanism for large attributes by using the streaming APIs for read and update access. These APIs stream the attribute content directly from the database server without caching in the client.

Server Side SQL Execution

The associative access performed by SQL queries is typically data intensive, therefore SQL queries and SQL methods are always executed on the server side and only the result sets are cached on the client side.

7 Running the Demo Applications

Demo applications are installed to illustrates diverse features of Matisse. For the .NET binding, the *Matisse .NET Programmer's Guide* explains how to run these applications.

Data Migration Demo

This demo is accessible through the "Read me" page in your Matisse installation. On Windows, click on "Matisse->Read me" from the Windows Start menu.

The demo called *President* shows how to create your application schema with SQL DDL (Data Definition Language), and populate your database with relational data trough the Matisse Data Transformation Services (DTS) facility and then query the database with SQL.

Reusable SQL Components Demo

This demo is also accessible through the "Read me" page in your Matisse installation. On Windows, click on "Matisse->Read me" from the Windows Start menu.

The demo called *GoalMind* is a simple CRM (Customer Relationship Management) application that illustrates how to build reusable SQL components with SQL Methods.

The second demo called *Friends* is a simple social networks application that illustrates how to implement a graph search algorithm to solve the single-source shortest path problem with SQL Methods.

XML Documents Demo

This demo is accessible through the "Read me" page in your Matisse installation. On Windows, click on "Matisse->Read me" from the Windows Start menu.

The demo called *Media* shows how to create your application schema with ODL (Object Definition Language), and import XML data into a Matisse database. The XML document imported into Matisse contains both text and multimedia data.

Data Reporting Demo

This demo is accessible through the "Read me" page in your Matisse installation. On Windows, click on "Matisse->Read me" from the Windows Start menu.

The demo called *Reports* shows how to create your application schema with ODL (Object Definition Language), and populate your database with XML Documents and then query the database with SQL to build ad-hoc reports.

Matisse Lite Edition Demo

This demo is accessible through the "Read me" page in your Matisse installation. On Windows, click on "Matisse->Read me" from the Windows Start menu.

The demo called *Lite* shows how to develop an application with Matisse Lite, the embedded version of Matisse DBMS. Matisse Lite is a compact library that implements the server-less version of Matisse. Matisse Lite provides a transactional, multi-user database engine self-contained in an application process.

.NET Demo Programs

The Matisse .NET demo programs come with the .NET binding installation. You can download the .NET binding with the ADO.NET data provider from http://www.matisse.com/developers/downloads/.

The Matisse .NET demo programs are provided both for C# and VB.NET. They illustrate:

- How to use ADO.NET

- How to create/update/delete objects

- How to manage relationships between objects

- How to use indexes or entry-point dictionaries to retrieve objects

- How to manage database connections, transactions and version access (read-only transaction)

- How to manipulate object from Matisse Class Reflection APIs

- How to manage database events

- How to manage a pool of connections

- How to extend Matisse Object Factories

- How to use Matisse Data Classes

- How to use object versioning

- How to generate source code for persistent classes in any .NET programming language

Java Demo Programs

The Matisse Java demo programs can downloaded with the *Matisse Java Programmer's Guide* from the Matisse Documentation web page http://www.matisse.com/developers/documentation/.

Requirements

- Install Sun Java Development Kit version 6 or later for your operating system.

- Set the MATISSE_HOME and JAVA_HOME environment variables to the top-level directories of the Matisse and JDK installations.

- If you are running Windows, append ;%matisse_home%\bin;%java_home%\bin to the PATH user variable.

Demos

The Java demo programs illustrate:

- How to use JDBC

- How to create/update/delete objects

- How to manage relationships between objects

- How to use indexes or entry-point dictionaries to retrieve objects

- How to manage database connections, transactions and version access (read-only transaction)

- How to manipulate object from Matisse Class Reflection APIs

- How to manage database events

- How to manage a pool of connections

- How to extend Matisse Object Factories

- How to use object versioning

C++ Demo Programs

The Matisse C++ demo programs can downloaded with the *Matisse C++ Programmer's Guide* from the Matisse Documentation web page http://www.matisse.com/developers/documentation/.

The C++ demo programs illustrate:

- How to create/update/delete objects

- How to manage relationships between objects

- How to use indexes or entry-point dictionaries to retrieve objects

- How to manage database connections, transactions and version access (read-only transaction)

- How to manipulate object from Matisse Class Reflection APIs

- How to manage database events

- How to use object versioning

8 Matisse Tools and Documentation

8.1 Development Tools

C API	A set of functions that allow user access to and manipulation of almost every aspect of Matisse databases. The C API functions are defined in the "include" file matisse.h.
	Documentation: the *Matisse C API Reference*.
C++ binding	The C++ binding (included in the standard Matisse installation) provides access to Matisse databases within C++ applications. Functions are defined in the "include" file matissecxx.h.
	Documentation: the *Matisse C++ Programmer's Guide* and the Doc++-generated API reference in the docs subdirectory of the Matisse program directory
.NET binding and ADO.NET	The .NET binding provides access to Matisse databases within .NET applications. It includes both the ADO.NET data provider and the native object interface. The binding is available on our web site at www.matisse.com/developers/downloads/
Eiffel binding	The Eiffel binding provides access to Matisse databases within Eiffel applications. You can download the binding and documentation from:
	www.matisse.com/developers/downloads/
Java binding	The Java binding (included in the standard Matisse installation) provides access to Matisse databases within Java applications. The Matisse Java methods are defined in matisse.jar in the lib subdirectory of the Matisse program directory.
	Documentation: the *Matisse Java Programmer's Guide* and the javadoc-generated API reference in the docs subdirectory of the Matisse installation directory.
ODBC driver	The Matisse SQL ODBC driver provides access to Matisse databases from ODBC-compliant tools.
	Documentation: the Matisse installation guides for Linux, Solaris, and MS Windows.
Perl, PHP, and Python bindings	These open-source bindings provide access to Matisse databases from the respective languages. You can download the bindings and documentation from:
	www.matisse.com/developers/downloads/ www.matisse.com/developers/documentation/

Smalltalk binding	The Smalltalk binding provides transparent access to Matisse databases from VisualWorks applications. You can download the binding from: www.matisse.com/developers/downloads/
Matisse SQL	The Matisse SQL language can be used interactively using the Matisse Enterprise Manager, or embedded in applications created with the Matisse C API or the language bindings. Documentation: the *Matisse SQL Programmer's Guide*, the *Matisse C API Reference*, and the online help in the mt_sql utility.
Matisse XML	Matisse XML can be used to exchange data with other applications or to transfer data between Matisse databases on different platforms (for example, Windows and Linux). Documentation: the *Matisse XML Programming Guide* and the online help in the mt_xml utility

8.2 Utilities

mt_emgr	Matisse Enterprise Manager. You may use this tool to create, configure, initialize, start, monitor, stop, back up, and restore databases, as well as to manage database disk usage, versions, and security. The administration operations can also be performed with the command line utilities mt_backup, mt_connection, mt_file, mt_partition, mt_replicate, mt_server, mt_transaction, mt_user and mt_version. Documentation: *Matisse Server Administration Guide*
mt_modeler	Matisse Database Modeler. You may use this tool to design and manage yout database schema. Documentation: *Matisse Modeler User's Guide*
mt_sdl	This command line utility allows you to import and export a database schema using SQL DDL (Data Definition Language) or ODL (Object Definition Language). It is also used to generate persistent class files for the C++, C#, Java, and Eiffel bindings. Documentation: *Matisse ODL Programmer's Guide and Matisse SQL Programmer's Guide*
mt_sql	Matisse SQL interactive query tool. Documentation: *Matisse SQL Programmer's Guide*

mt_xml

Matisse XML import/export utility. With this tool you can load data from an XML file into a database, dump data from a database to an XML file, and transfer data between Matisse databases on different platforms (for example, Windows and Solaris).

Documentation: *Matisse XML Programming Guide*

mt_dts

Matisse Data Transaformation Services utility. With this tool you can extract, transform and load data from a relational source into a Matisse database.

Documentation: *Matisse Data Transaformation Services Guide*

mt_xsr

Matisse XML-based Snapshot Replication utility. With this tool you can replicate databases.

Documentation: *Matisse Server Administration Guide*

9   Schema Objects Properties

This section details the definition of the schema objects that make up your application schema. A schema is made of a set of classes, attributes, relationships, methods, indexes, and/or entry-point dictionaries.

9.1 Rules for Naming Schema Objects

• The names that can be used for all schema objects must contain only alphanumeric characters and the underscore character (a-z, A-Z, 0-9, and _), and must start with a letter.

• Names must not contain spaces or punctuation other than the underscore character.

So, for example, Class_9 is a valid name, but Class 9, Class-9, Class.9, and 9Class are invalid.

9.2 Class Properties

Matisse schema classes are always persistent. A class has the following properties:

class name	Required. Class names must be unique.
superclass name	Optional. If a superclass is specified, this class inherits all of its properties. Multiple superclasses may be specified if all the programming languages you are using support multiple inheritance (C# and Java do not).
attributes	Optional. Define the values to be stored in instances of this class (and any subclasses).
relationships	Optional. Define relationships between instances of this class (and any subclasses) and instances of the successor class (or its subclasses).
index specifications	Optional.
entry-point dictionary specifications	Optional.
method specifications	Optional. The SQL methods for the class.

For instance the definition of the class Movie with the attributes Title and Rating, and the associated method CountMovies is exported in DDL format like this:

```
--
-- Class Definitions
--
CREATE TABLE Movie (
    Title VARCHAR(64),
    Rating VARCHAR(4)
```

```
                );

                --
                -- Methods on Movie
                --
        CREATE STATIC METHOD CountMovies(aRating STRING)
                RETURNS INTEGER
                FOR Movie
                BEGIN
                    DECLARE Cnt INTEGER DEFAULT 0;
                    SELECT COUNT(*) INTO Cnt FROM Movie WHERE Rating = aRating;
                    RETURN Cnt;
                END;
```

9.3 Attribute Properties

An attribute has the following properties:

attribute name Attribute names must be unique within a class and any of its subclasses. In other words, two classes may have attributes of the same name, unless one class inherits from the other.

type The data type (STRING, INTEGER, etc.) of the attribute.

nullable A boolean indicating whether the attribute will allow NULL values. Note that when creating a schema with ODL attributes *are not* nullable unless you specify Nullable, while in SQL DDL attributes *are* nullable unless you specify NOT NULL.

When nullable is True and no default value is specified, Matisse automatically sets the default value to NULL.

When nullable is False and no default value is specified, when you create an instance you must explicitly set the attribute value before attempting to commit your updates.

maximum size The maximum size for variable size attributes. For instance with SQL DDL, a maximum size of 12 for a string type attribute can be expressed as 'VARCHAR(12)'.

default value Optional. A value returned when accessing an attribute value that has not been set or that has been cleared.

Note that default values are not physically *stored in* the corresponding attribute values of instances instantiated from the class. Thus if you change the default value for an attribute in your schema, it will be immediately reflected in the associated instances. To put it another way, in Matisse default values are class variables, not initial values.

9.4 Relationship Properties

Relationships are usually declared in pairs. For example, the Employee class might specify a MemberOf relationship with the Department class, and the Department class a corresponding Members relationship with the Employee class. With the Matisse Modeler tool this relationship pair is represented as a single association, and the individual relationships as "roles."

A relationship has the following properties:

relationship name	Relationship names must be unique within a class and any of its subclasses.
successor class	The "target" class for this relationship. In the example below, Department is the successor class of MembersOf, and Employee is the successor class of Members.
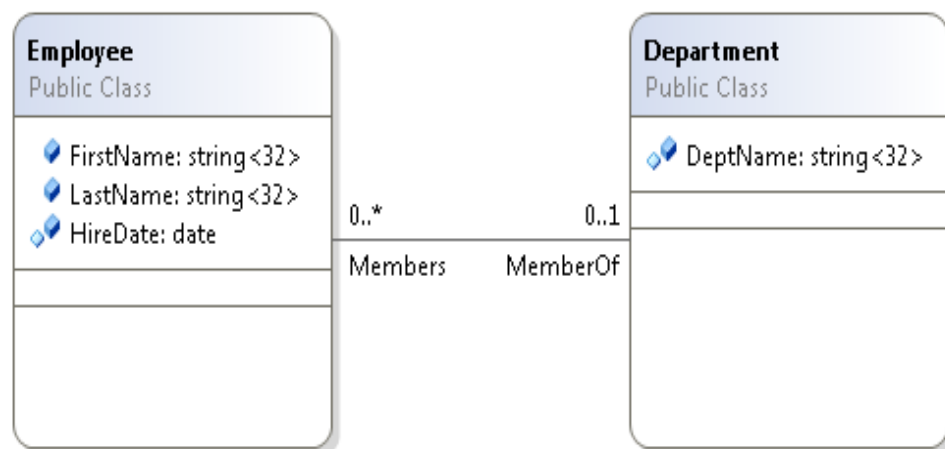
	A class may be the successor class of its own relationships. A typical example is the Father / Children relationship pair between instances of a class Person.
inverse relationship	The other relationship of the pair, specified in the successor class.
	If appropriate a relationship may be specified as its own inverse, a typical example is the Spouse relationship between instances of a class Person.
cardinality constraint	The minimum and maximum number of successor instances allowed. The minimum cardinality is usually 0 or 1, indicating whether successors are optional or required. The maximum cardinality is usually 1 or -1 (representing unlimited), indicating whether it is a "to-one" or "to-many" relationship. Common cardinality settings (and their UML counterparts used in schema diagrams) include:

0, -1: may have any number of successors, or none (UML *)

0, 1: may have one successor, or none (UML 0..1)

1, 1: must have one and only one successor (UML 1)

1, -1: must have at least one successor (UML 1..*)

read-only constraint	To simplify programming, one of the two relationships in a pair is usually set read-only, so that all updates must be performed from the other side.

- In SQL DDL, this is done by specifying the readonly parameter.

- In the Matisse Modeler this is set by the Navigable option on the role's Details tab. By default, Role A is read-write (Navigable checked) and Role B is read-only (Navigable unchecked).

9.5 Index Properties

An index is declared for a given class, it has the following properties:

index name	Index names must be unique within a class and any of its subclasses.
criteria	The names of the attribute(s) to be indexed. You may specify up to four criteria.
sort order	Set separately for each criterion. Specifies whether the index will be sorted ascending or descending for that criterion.
unique key constraint	If set to True, each entry in the index must be unique. If set to False, the index may contain duplicate entries.

The key size for an index is limited to 256 bytes. For an index defined with multiple criteria, the sum of the sizes for the attributes to be indexed must not exceed 256.

9.6 Entry-Point Dictionary Properties

An entry-point dictionary has the following properties:

dictionary name	Entry-point dictionary names must be unique within a class and any of its subclasses.
attribute name	The name of the attribute for which the entry-point dictionary will be maintained.
unique key constraint	If set to True, each entry in the dictionary must be unique. If set to False, the dictionary may contain duplicate entries.
case sensitivity	If set to True, dictionary lookups are case-sensitive. If set to False, lookups are case-insensitive.
make-entry function	The name of the function to be used to generate keys (also called entry-points) for the dictionary. Two such functions are included with Matisse: "make-entry" generates one key per instance based on the full attribute value, "make-full-text-entry" extracts the words contained in a textual attribute value and generates a separate key for each distinct word in the attribute value.

Glossary

abstract class: A class which cannot be instantiated, that is, a class from which other classes may be derived, but which cannot be used to create objects. (Matisse itself has no abstract classes; the term is used only in language-binding documentation.)

atomic transaction: A transaction that either successfully applies all of its updates to the database or leaves the database unchanged.

attribute value: The value that is stored in a database object for a given attribute.

attribute: A property of a class that defines the values that will be stored in objects instantiated from the class. For example, an object representing a person would typically contain some values for the person's first name and last name.

cardinality: A constraint for a relationship that defines the minimum and maximum number of successors allowed.

class: A schema object that defines the structure of the objects that can be generated from the class. A class definition can contains *attribute* and *relationship* properties, superclasses, indexes and entry-point dictionaries. See also *abstract class*, *metaclass*.

class method: A method that applies to a class as a whole rather than to individual objects instantiated from the class; for example, a method that returns the number of instances of a class.

class variable: A value shared by all instances of a class. In Matisse, class variables can be defined by modifying or extending MtClass (see *meta-schema*)

constructor: A class method that creates an object. It is a convention in some object-oriented programming languages that every class has a constructor with the same name as the class.

data type: See *type*.

database: The live database running on a Matisse server.

database schema: See *schema*.

database server: The process that runs a Matisse database.

datafile: A file, disk partition (raw device) or ramdisk device allocated to a Matisse database. A database can have up to 255 datafiles on as many disks. For fault tolerance and best performance, in a production environment each datafile should be on a different disk.

datapage: The minimum amount of data that can be read from or written to disk by the Matisse server in any single I/O operation. Datapages are contained in datafiles.

default value: A value returned when Matisse attempts to read an attribute value that has either not been set or that has been removed from an object. Note that this is not an initial value copied to objects but rather a class variable associated with the attribute descriptor; see *default value* on page 29 for a detailed discussion.

descriptor: In Matisse schema definition elements like classes, attributes and relationships are stored as objects in the database and are sometimes referred to as descriptors. For instance an attribute descriptor describes the name and the type of value that an attribute can accept within a data object.

entry-point dictionary: An indexing structure that allows fast retrieval of objects based on keyword searches. It is typically used for full-text indexing or to provide a fast object lookup capability. When creating an entry-point dictionary, you associate a make-entry function that will be used to automatically generate the keywords to be indexed.

garbage collection: Permanently removing obsolete object versions from the database in order to recover disk space.

identifier: See *object identifier*.

index: An indexing structure that optimizes retrieval of objects based on a value or set of values and provides fast execution for ORDER BY SQL statements.

instance: See *object*.

interface: In ODL, a set of statements that declares a class. In Matisse API and language binding references, it may have other meanings specific to the language in question.

inverse relationship: All relationships are declared with both direct and inverse orientations. Which is which depends on which end of the relationship you start from. For example, in the schema diagrammed below, Department.Members is the inverse of the direct relationship Employee.MemberOf, which itself is the inverse of the direct relationship Department.Members.
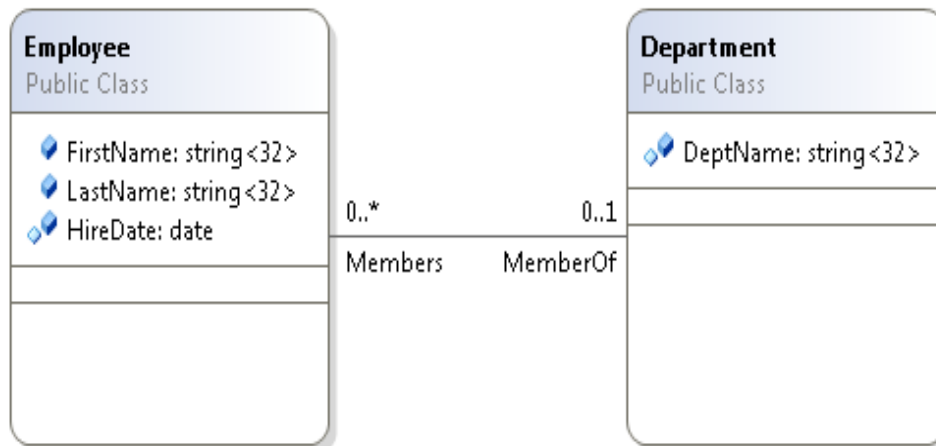
In some cases, a relationship may be its own inverse (for instance a Spouse relationship).

list of successors: See *relationship successors*.

literal: A *literal* is a specific symbol that represents a data value. For example, the literals 2.6 and 2.60 both represent the same floating-point value.

make-entry function: See *entry-point dictionary*.

metaclass: A class whose instances are classes, rather than data objects. The only metaclass in Matisse is MtClass.

meta-schema: The initial set of Matisse classes from which classes and other *schema objects* are instantiated. For instance MtAttribute, MtClass, MtEntryPointDictionary, MtIndex, and MtRelationship are part of the meta-schema.

method: A function defined for a class. Procedural components of Matisse database applications are typically implemented using methods.

null: An undefined value. An attribute value is set as undefined by setting it to the value NULL, which is possible only if nullable has been set in the corresponding attribute descriptor. For discussion of some consequences of making an attribute non-nullable, see *default value*.

object: In Matisse documentation, used by itself, it generally refers to an instance of a class, that is, a persistent object. An object contains *attribute values* and *relationship successors* corresponding to each of the *attributes* and *relationships* defined for its class in the *schema*. See also *schema object*.

Object Description Language (ODL): A language that can be used to define the schema of a Matisse database and to generate persistent classes for the different language bindings. Matisse ODL follows the ODL standard promulgated by the Object Data Management Group (odmg.org).

object identifier: All objects in a Matisse database are automatically assigned stable unique identifiers that the system uses to distinguish them and to manage relationships.

In C, C++, Java, and other languages, "identifier" commonly refers to the name of an object or stored value. To avoid confusion, it is not used in that sense in the Matisse documentation.

ODL: See *Object Description Language (ODL)*.

OID: See *object identifier*.

predecessor: An object that references other objects through a given *relationship* is said to be the predecessor of the objects that it references.

programming type: In reference to the C API, a C type (as opposed to a Matisse *type*).

property: An attribute or a relationship defined for a class. For example, the attribute LastName is a property of the class Person.

relationship: Relationships are class properties that define links between objects. A relationship is defined between two (or more) classes. For instance the relationship Department.Members may be defined between the class Department and the class Employee. In this example the class Employee is referred to as the successor class. See also *inverse relationship*.

relationship successors: The object content corresponding to a *relationship* defined in the schema class for the object. The relationship part of an object stores the list of object identifiers that identify its successors (also called the "list of successors").

When an object is deleted, Matisse automatically removes its object identifier from the relationship part of the objects to which it is a successor.

savetime: See *versioning*.

scalar type: A type that takes a single value (as opposed to a list type).

schema: The portion of a database that describes the structure of the data objects. Also called an application schema to distinguish it from the *meta-schema*. For example, a schema might contain classes such as Employee and Department, from which data objects representing specific individuals and departments can be instantiated; *attributes* such as Employee.Name, Employee.SSNum, and Department.Name which define the values to be stored in the data objects; and *relationships* such as Employee.MemberOf and Department.Members define the possible links between the data objects of the Employee and Department classes. See also *meta-schema*.

schema object: An object that describes an element of a *schema* in a Matisse database. For instance the definition of the class Employee is stored as a schema object. See also *schema*.

static: Describes a method or other function associated with a class as a whole, not with particular instances of a class: for example, a function that returns the number of objects in a class. See *class variable*.

stream: A byte-stream of data sent from the Matisse server to a client, or vice-versa.

successor: An object reachable by navigational access through the *relationship successors* of another object.

successor class: See *relationship*.

superclass: A class that is inherited by another class.

transaction: In Matisse, the term "transaction" generally means a read-write transaction. A read-only transaction is called a "version access."

type: In regard to the definition of an attribute, short for *data type*, it specifies which type of data (boolean, integer, string, etc.) the associated values will store.

UML: Unified Modeling Language. See www.omg.org/technology/uml for more information.

user method: In a schema class generated for a language binding, a method added manually by the user, rather than generated automatically by the mt_sdl utility.

value: See *attribute value*.

version: A copy of an object at a given time, the latest copy being called the "current version". Also used as a synonym for "savetime," it provides a snapshot of an entire database at a given time. See *versioning*.

version access: A read-only transaction that provides consistent access to a snapshot of the database without locking.

version collection: See *garbage collection*.

versioning: The automatic preservation of old object versions in order to preserve a consistent "snapshot" of the database at a particular time (a "savetime"). When an object is modified, Matisse saves the changes in a new copy of the object. The unmodified version of the object is preserved until all the relevant version accesses have been closed.

Named versions may also be created manually with the mt_dba utility or the mt_version command, or by specifying a version name as an argument to transaction commit.

Unnamed versions created by Matisse are removed automatically during garbage collection. Named versions created by users or client applications are kept until they are explicitly undeclared.