Ada Database Objects Programmer's Guide

STEPHANE CARREZ

2022-08-20

Contents

1 Introduction **4**

2 Installation **5**
2.1 Before Building . . . 5
2.2 Database Driver Installation . . . 5
2.2.1 Ubuntu . . . 5
2.2.2 Windows . . . 5
2.3 Configuration . . . 6
2.4 Build . . . 7
2.5 Installation . . . 7
2.6 Using . . . 8

3 Tutorial **9**
3.1 Defining the data model . . . 9
3.2 Generating the Ada model and SQL schema . . . 12
3.3 Getting a Database Connection . . . 12
3.4 Opening a Session . . . 13
3.5 Creating a database record . . . 13
3.6 Loading a database record . . . 14
3.7 Getting a list of objects . . . 15
3.8 Running SQL queries . . . 15

4 Session **17**
4.1 Database Drivers . . . 17
4.1.1 MySQL Database Driver . . . 18
4.1.2 SQLite Database Driver . . . 19
4.1.3 PostgreSQL Database Driver . . . 19
4.2 Connection string . . . 20
4.3 Session Factory . . . 20
4.4 Database Caches . . . 21

5 Database Statements **22**
5.1 Query Parameters . . . 22
5.1.1 Parameter Expander . . . 23
5.2 Query Statements . . . 23
5.3 Named Queries . . . 23
5.3.1 XML Query File . . . 24

5.3.2 SQL Result Mapping 24
5.3.3 SQL Queries 25
5.3.4 Using Named Queries 26

6 Model Mapping **28**
6.1 Table definition 28
6.2 Column mapping 29
6.3 Primary keys 31
6.4 Relations 32
6.5 Versions 32
6.6 Loading Objects 34
6.7 Modifying Objects 35
6.8 Deleting Objects 35
6.9 Sequence Generators 36
6.9.1 HiLo Sequence Generator 36

7 Database schema migration **37**
7.1 Migration mechanism 37
7.2 Taking into account the migration during development 39

8 Troubleshooting **40**
8.1 Change the log configuration 40
8.2 Handling exceptions 41

1 Introduction

The Ada Database Objects is an Object Relational Mapping for the Ada05 programming language. It allows to map database objects into Ada records and access database content easily. The library supports PostgreSQL, MySQL, SQLite as databases. Most of the concepts developed for ADO come from the Java Hibernate ORM.

The ORM uses either an XML mapping file, a YAML file or an UML model, a code generator and a runtime library for the implementation. It provides a database driver for PostgreSQL, MySQL and SQLite. The ORM helps your application by providing a mapping of your database tables directly in the target programming language: Ada05 in our case. The development process is the following:

- You design your database model either using a UML tool or by writing an XML or YAML description file,
- You generate the Ada05 mapping files by using the Dynamo code generator,
- You generate the SQL database tables by using the same tool,
- You write your application on top of the generated code that gives you direct and simplified access to your database.

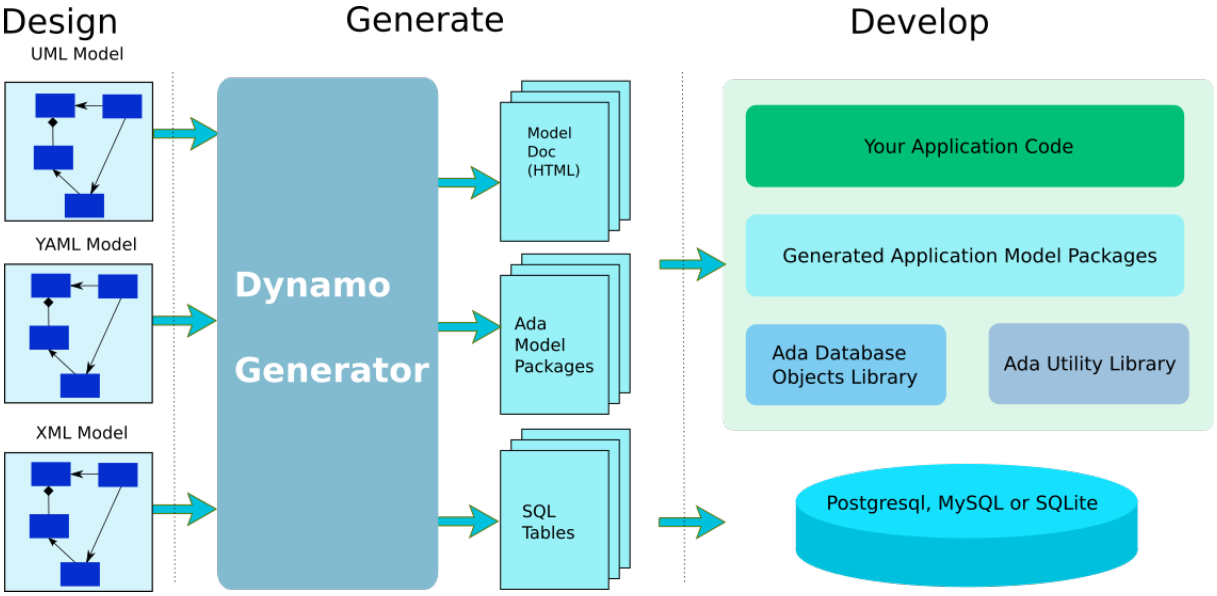

Figure 1: *ORM Development Model*

This document describes how to build the library and how you can use the different features to simplify and help you access databases from your Ada application.

2 Installation

This chapter explains how to build and install the library.

2.1 Before Building

Before building ADO, you will need:

- Ada Utility Library
- XML/Ada
- Either the PostgreSQL, MySQL or SQLite development headers installed.

First get, build and install the XML/Ada and then get, build and install the Ada Utility Library.

2.2 Database Driver Installation

The PostgreSQL, MySQL and SQLite development headers and runtime are necessary for building the ADO driver. The configure script will use them to enable the ADO drivers. The configure script will fail if it does not find any database driver.

2.2.1 Ubuntu

MySQL Development installation

```
1 sudo apt-get install libmysqlclient-dev
```

MariaDB Development installation

```
1 sudo apt-get install mariadb-client libmariadb-client-lgpl-dev
```

SQLite Development installation

```
1 sudo apt-get install libsqlite3-dev
```

PostgreSQL Development installation

```
1 sudo apt-get install postgresql-client libpq-dev
```

2.2.2 Windows

It is recommended to use msys2 available at https://www.msys2.org/ and use the `pacman` command to install the required packages.

```
pacman -S git
pacman -S make
pacman -S unzip
pacman -S base-devel --needed
pacman -S mingw-w64-x86_64-sqlite3
```

For Windows, the installation is a little bit more complex and manual. You may either download the files from MySQL and SQLite download sites or you may use the files provided by Ada Database Objects in the `win32` directory.

For Windows 32-bit, extract the files:

```
cd win32 && unzip sqlite-dll-win32-x86-3290000.zip
```

For Windows 64-bit, extract the files:

```
cd win32 && unzip sqlite-dll-win64-x64-3290000.zip
```

If your GNAT 2019 compiler is installed in `C:/GNAT/2019`, you may install the liblzma, MySQL and SQLite libraries by using msys cp with:

```
cp win32/*.dll C:/GNAT/2019/bin
cp win32/*.dll C:/GNAT/2019/lib
cp win32/*.lib C:/GNAT/2019/lib
cp win32/*.a C:/GNAT/2019/lib
```

2.3 Configuration

The library uses the `configure` script to detect the build environment, check which databases are available and configure everything before building. If some component is missing, the `configure` script will report an error. The `configure` script provides several standard options and you may use:

- `--prefix=DIR` to control the installation directory,
- `--with-mysql=PATH` to control the path where `mysql_config` is installed,
- `--with-ada-util=PATH` to control the installation path of Ada Utility Library,
- `--enable-mysql` to enable the support for MySQL,
- `--enable-postgresql` to enable the support for PostgreSQL,
- `--enable-sqlite` to enable the support for SQLite,
- `--enable-shared` to enable the build of shared libraries,
- `--disable-static` to disable the build of static libraries,
- `--enable-distrib` to build for a distribution and strip symbols,

- `--disable-distrib` to build with debugging support,
- `--enable-coverage` to build with code coverage support (`-fprofile-arcs -ftest-coverage`),
- `--help` to get a detailed list of supported options.

In most cases you will configure with the following command:

```
./configure
```

2.4 Build

After configuration is successful, you can build the library by running:

```
make
```

After building, it is good practice to run the unit tests before installing the library. The unit tests are built and executed using:

```
make test
```

And unit tests are executed by running the `bin/ado_harness` test program. A configuration file is necessary to control the test parameters including the test database to be used. To run the tests with a MySQL database, use the following command:

```
bin/ado_harness -config test-mysql.properties
```

and with a SQLite database, use the following command:

```
bin/ado_harness -config test-sqlite.properties
```

2.5 Installation

The installation is done by running the `install` target:

```
make install
```

If you want to install on a specific place, you can change the `prefix` and indicate the installation direction as follows:

```
make install prefix=/opt
```

2.6 Using

To use the library in an Ada project, add the following line at the beginning of your GNAT project file:

```
with "ado";
with "ado_all";
```

It is possible to use only a specific database driver, in that case your GNAT project file could be defined as follows:

```
with "ado";
with "ado_mysql";
with "ado_sqlite";
with "ado_postgresql";
```

where the `ado_mysql`, `ado_sqlite` and `ado_postgresql` are optional and included according to your needs.

3 Tutorial

This small tutorial explains how an application can access a database (PostgreSQL, MySQL or SQLite) to store its data by using the Ada Database Objects framework. The framework has several similarities with the excellent Hibernate Java framework.

The ADO framework is composed of:

- A code generator provided by Dynamo,
- A core runtime,
- A set of database drivers (PostgreSQL, MySQL, SQLite).

The tutorial application is a simple user management database which has only one table.

3.1 Defining the data model

The first step is to design the data model. You have the choice with:

- Using an UML modeling tool such as ArgoUML,
- Writing an XML file following the Hibernate description,
- Writing a YAML description according to the Doctrine mapping.

In all cases, the model describes the data table as well as how the different columns are mapped to an Ada type. The model can also describe the relations between tables. XML and YAML data model files should be stored in the `db` directory.

Let's define a mapping for a simple `user` table and save it in `db/user.hbm.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-mapping default-cascade="none">
    <class name="Samples.User.Model.User"
           table="user" dynamic-insert="true" dynamic-update="true">
        <comment>Record representing a user</comment>
        <id name="id" type="ADO.Identifier" unsaved-value="0">
            <comment>the user identifier</comment>
            <column name="id" not-null="true" unique="true" sql-type="
               BIGINT"/>
            <generator class="sequence"/>
        </id>
        <version name="version" type="int" column="object_version" not-
           null="true"/>
        <property name="name" type="String">
            <comment>the user name</comment>
            <column name="name" not-null="true" unique="false" sql-type
               ="VARCHAR(256)"/>
```

```
15          </property>
16          <property name="email" type="String" unique='true'>
17              <comment>the user email</comment>
18              <column name="email" not-null="true" unique="false" sql-
                    type="VARCHAR(256)"/>
19          </property>
20          <property name="date" type="String">
21              <comment>the user registration date</comment>
22              <column name="date" not-null="true" unique="false" sql-type
                    ="VARCHAR(256)"/>
23          </property>
24          <property name="description" type="String">
25              <comment>the user description</comment>
26              <column name="description" not-null="true" unique="false"
                    sql-type="VARCHAR(256)"/>
27          </property>
28          <property name="status" type="Integer">
29              <comment>the user status</comment>
30              <column name="status" not-null="true" unique="false" sql-
                    type="Integer"/>
31          </property>
32      </class>
33  </hibernate-mapping>
```

The YAML description is sometimes easier to understand and write and the content could be saved in the `db/users.yaml` file.

```
1  Samples.User.Model.User:
2    type: entity
3    table: user
4    description: Record representing a user
5    hasList: true
6    indexes:
7    id:
8      id:
9        type: identifier
10       column: id
11       not-null: true
12       unique: true
13       description: the user identifier
14   fields:
15     version:
16       type: integer
```

```
      column: object_version
      not-null: true
      version: true
      unique: false
      description:
    name:
      type: string
      length: 255
      column: name
      not-null: true
      unique: false
      description: the user name
    email:
      type: string
      length: 255
      column: email
      not-null: true
      unique: false
      description: the user email
    date:
      type: string
      length: 255
      column: date
      not-null: true
      unique: false
      description: the user registration date
    description:
      type: string
      length: 255
      column: description
      not-null: true
      unique: false
      description: the user description
    status:
      type: integer
      column: status
      not-null: true
      unique: false
      description: the user status
```

These XML and YAML mapping indicate that the database table `user` is represented by the `User` tagged record declared in the `Samples.User.Model` package. The table contains a `name`, `description`,

`email` and a `date` column members which are a string. It also has a `status` column which is an integer. The table primary key is represented by the `id` column. The `version` column is a special column used by the optimistic locking.

3.2 Generating the Ada model and SQL schema

The Dynamo code generator is then used to generate the package and Ada records that represent our data model. The generator also generates the database SQL schema so that tables can be created easily in the database.

```
dynamo generate db
```

The generator will build the package specification and body for `Samples.User.Model` package. The files are created in `src/model` to make it clear that these files are model files that are generated. The database table `user` is represented by the Ada tagged record `User_Ref`. The record members are not visible and to access the attributes it is necessary to use getter or setter operations.

The SQL files are generated for every supported database in the `db/mysql`, `db/sqlite` and `db/postgresql` directories. The generator generates two SQL files in each directory:

- A first SQL file that allows to create the tables in the database. The file name uses the pattern `create-`*name-driver*.
- A second SQL file that contains `DROP` statements to erase the database tables. The file name uses the pattern `drop-`*name-driver*.

When you modify the UML, XML or YAML model files, you should generate again the Ada and SQL files. Even though these files can be generated, it is recommended to store these generated files in a versioning systems such as `git` because this helps significantly in tracking changes in the data model.

3.3 Getting a Database Connection

To access the database, we will need a database connection. These connections are obtained from a factory and they are represented by a session object.

The session factory is the entry point to obtain a database session.

```
with ADO.Sessions;
with ADO.Sessions.Factory;
   ...
   Factory : ADO.Sessions.Factory.Session_Factory;
```

The factory can be initialized by giving a URI string that identifies the driver and the information to connect to the database. Once created, the factory returns a session object to connect to that database. To connect to another database, another factory is necessary.

To get access to a MySQL database, the factory could be initialized as follows:

```
1    ADO.Sessions.Factory.Create (Factory, "mysql://localhost:3306/
        ado_test?user=test");
```

And to use an SQLite database, you could use:

```
1    ADO.Sessions.Factory.Create (Factory, "sqlite:///tests.db");
```

For a PostgreSQL database, the factory would look like:

```
1    ADO.Sessions.Factory.Create (Factory, "postgresql://localhost:5432/
        ado_test?user=test");
```

Factory initialization is done once when an application starts. The same factory object can be used by multiple tasks.

3.4 Opening a Session

The session is created by using the `Get_Session` or the `Get_Master_Session` function of the factory. Both function return a session object associated with a database connection. The `Get_Session` will return a `Session` object which is intended to provide read-only access to the database. The `Get_Master_session` returns a `Master_Session` object which provides a read-write access to the database.

In a typical MySQL Master/Slave replication, the `Master_Session` will refer to a connection to the MySQL master while the `Session` will refer to a slave. With an SQLite database, both sessions will in fact share the same SQLite internal connection.

To load or save the user object in the database, we need a `Master_Session` database connection:

```
1 with ADO.Sessions;
2 ...
3   Session : ADO.Sessions.Master_Session := Factory.Get_Master_Session;
```

3.5 Creating a database record

To create our first database record, we will declare a variable that will represent the new database record. The `User_Ref` represents a reference to such record.

```
with Samples.User.Model;
   ...
   User : Samples.User.Model.User_Ref
```

After this declaration, the variable does not refer to any database record but we can still set some fields:

```
   User.Set_Name ("Harry");
   User.Set_Age (17);
```

To save the object in the database, we just need to call the `Save` operation. To save the object, we need a database session that is capable of updating and inserting new rows. If the object does not yet have a primary key, and if the primary key allocation mode is set to `hilo`, the ADO runtime will allocate a new unique primary key before inserting the new row.

```
   User.Save (Session);
```

The primary key can be obtained after the first `Save` with the following operation:

```
   Id : ADO.Identifier := User.Get_Id;
```

3.6 Loading a database record

Loading a database record is quite easy and the ADO framework proposes two mechanisms. First, let's declare our user variable:

```
   Harry : User_Ref;
```

Then we can load the user record from the primary key identifier (assuming the identifier is "'23"'):

```
   Harry.Load (Session, 23);
```

If the user cannot be found, the `Load` operation will raise the `NOT_FOUND` exception.

In many cases, we may not know the primary key but a search on one or several columns may be necessary. For this, we can create a query filter and use the `Find` operation. To use a query filter, we need first to declare a `Query` object:

```
with ADO.SQL;
...
   Query : ADO.SQL.Query;
```

On the query object, we have to define the filter which represents the condition and set the possible parameters used by the filter.

```
1    Query.Bind_Param (1, "Harry");
2    Query.Set_Filter ("name = ?");
```

Once the query is defined and initialized, we can find the database record:

```
1    Found : Boolean;
2  ...
3    User.Find (Session, Query, Found);
```

Unlike the `Load` operation, `Find` does not raise an exception but instead returns a boolean value telling whether the record was found or not. The database query (and filter) has to return exactly one record to consider the object as found.

3.7 Getting a list of objects

When several records have to be read, it is necessary to use the `List` operation together with a vector object.

```
1    Users : User_Vector;
```

The `List` operation gets the vector object, the database session and the query filter. If the vector contained some elements, they are removed and replaced by the query result.

```
1    List (Users, Session, Query);
```

3.8 Running SQL queries

Sometimes it is necessary to execute SQL queries to be able to get the result without having it to be mapped to an Ada record. For this, we are going to use the `ADO.Statements`

```
1  with ADO.Statements;
2     ...
3     Statement : ADO.Statements.Query_Statement := Session.
         Create_Statement ("SELECT COUNT(*) FROM user");
```

and then execute it and retrieve the result.

```
1     Statement.Execute;
```

```
2     if not Statement.Has_Elements then
3        Put_Line ("SQL count() failed")
4     else
5        Put_Line (Integer'Image (Statement.Get_Integer (0)));
6     end if;
```

4 Session

The `ADO.Sessions` package defines the control and management of database sessions. The database session is represented by the `Session` or `Master_Session` types. It provides operation to create a database statement that can be executed. The `Session` type is used to represent read-only database sessions. It provides operations to query the database but it does not allow to update or delete content. The `Master_Session` type extends the `Session` type to provide write access and it provides operations to get update or delete statements. The differentiation between the two sessions is provided for the support of database replications with databases such as MySQL.

4.1 Database Drivers

Database drivers provide operations to access the database. These operations are specific to the database type and the `ADO.Drivers` package among others provide an abstraction that allows to make the different databases look like they have almost the same interface.

A database driver exists for SQLite, MySQL and PostgreSQL. The driver is either statically linked to the application or it can be loaded dynamically if it was built as a shared library. For a dynamic load, the driver shared library name must be prefixed by `libada_ado_`. For example, for a `mysql` driver, the shared library name is `libada_ado_mysql.so`.

Driver name	Database
mysql	MySQL, MariaDB
sqlite	SQLite
postgresql	PostgreSQL

The database drivers are initialized automatically but in some cases, you may want to control some database driver configuration parameter. In that case, the initialization must be done only once before creating a session factory and getting a database connection. The initialization can be made using a property file which contains the configuration for the database drivers and the database connection properties. For such initialization, you will have to call one of the `Initialize` operation from the `ADO.Drivers` package.

```
1  ADO.Drivers.Initialize ("db.properties");
```

The set of configuration properties can be set programatically and passed to the `Initialize` operation.

```
Config : Util.Properties.Manager;
...
  Config.Set ("ado.database", "sqlite:///mydatabase.db");
  Config.Set ("ado.queries.path", ".;db");
  ADO.Drivers.Initialize (Config);
```

Once initialized, a configuration property can be retrieved by using the `Get_Config` operation.

```
URI : constant String := ADO.Drivers.Get_Config ("ado.database");
```

Dynamic loading of database drivers is disabled by default for security reasons and it can be enabled by setting the following property in the configuration file:

```
ado.drivers.load=true
```

Dynamic loading is triggered when a database connection string refers to a database driver which is not known.

4.1.1 MySQL Database Driver

The MySQL database driver can be initialize explicitly by using the `ado_mysql` GNAT project and calling the initialization procedure.

```
ADO.Mysql.Initialize ("db.properties");
```

The set of configuration properties can be set programatically and passed to the `Initialize` operation.

```
Config : Util.Properties.Manager;
...
  Config.Set ("ado.database", "mysql://localhost:3306/ado_test");
  Config.Set ("ado.queries.path", ".;db");
  ADO.Mysql.Initialize (Config);
```

The MySQL database driver supports the following properties:

Name	Description
user	The user name to connect to the server
password	The user password to connect to the server

Name	Description
socket	The optional Unix socket path for a Unix socket base connection
encoding	The encoding to be used for the connection (ex: UTF-8)

4.1.2 SQLite Database Driver

The SQLite database driver can be initialize explicitly by using the `ado_mysql` GNAT project and calling the initialization procedure.

```
1 ADO.Sqlite.Initialize ("db.properties");
```

The set of configuration properties can be set programatically and passed to the `Initialize` operation.

```
1 Config : Util.Properties.Manager;
2 ...
3   Config.Set ("ado.database", "sqlite:///regtests.db?synchronous=OFF&
       encoding=UTF-8");
4   Config.Set ("ado.queries.path", ".;db");
5   ADO.Sqlite.Initialize (Config);
```

The SQLite database driver will pass all the properties as SQLite `pragma` allowing the configuration of the SQLite database.

4.1.3 PostgreSQL Database Driver

The PostgreSQL database driver can be initialize explicitly by using the `ado_mysql` GNAT project and calling the initialization procedure.

```
1 ADO.Postgresql.Initialize ("db.properties");
```

The set of configuration properties can be set programatically and passed to the `Initialize` operation.

```
1 Config : Util.Properties.Manager;
2 ...
3   Config.Set ("ado.database", "postgresql://localhost:5432/ado_test?
       user=ado&password=ado");
4   Config.Set ("ado.queries.path", ".;db");
5   ADO.Postgresql.Initialize (Config);
```

The PostgreSQL database driver supports the following properties:

Name	Description
user	The user name to connect to the server
password	The user password to connect to the server

4.2 Connection string

The database connection string is an URI that specifies the database driver to use as well as the information for the database driver to connect to the database. The driver connection is a string of the form:

```
driver://[host][:port]/[database][?property1][=value1]...
```

The database connection string is passed to the session factory that maintains connections to the database (see ADO.Sessions.Factory).

4.3 Session Factory

The session factory is the entry point to obtain a database session. The `ADO.Sessions.Factory` package defines the factory for creating sessions.

```
with ADO.Sessions.Factory;
...
Sess_Factory : ADO.Sessions.Factory;
```

The session factory can be initialized by using the `Create` operation and by giving a URI string that identifies the driver and the information to connect to the database. The session factory is created only once when the application starts.

```
ADO.Sessions.Factory.Create (Sess_Factory, "mysql://localhost:3306/
   ado_test?user=test");
```

Having a session factory, one can get a database by using the `Get_Session` or `Get_Master_Session` function. Each time this operation is called, a new session is returned. The session is released when the session variable is finalized.

```
DB : ADO.Sessions.Session := Sess_Factory.Get_Session;
```

The session factory is also responsible for maintaining some data that is shared by all the database connections. This includes:

- the sequence generators used to allocate unique identifiers for database tables,
- the entity cache,
- some application specific global cache.

4.4 Database Caches

The ADO cache manager allows to create and maintain cache of values and use the cache from the SQL expander to replace cached values before evaluating the SQL. The SQL expander identifies constructs as follows:

```
1  $cache_name[entry-name]
```

and look for the cache identified by cache_name and then replace the cache entry registered with the name entry-name.

The cache manager is represented by the Cache_Manager type and the database session contains one cache manager. Applications may use their own cache in that case they will declare their cache as follows:

```
1   M : ADO.Caches.Cache_Manager;
```

A cache group is identified by a unique name and is represented by the Cache_Type base class. The cache group instance is registered in the cache manager by using the Add_Cache operation.

5 Database Statements

The `ADO.Statements` package provides high level operations to construct database statements and execute them. They allow to represent SQL statements (prepared or not) and provide support to execute them and retreive their result. The SQL statements are represented by several Ada types depending on their behavior:

- The `Statement` type represents the base type for all the SQL statements.
- The `Query_Statement` type is intended to be used for database query statements and provides additional operations to retrieve results.
- The `Update_Statement` type targets the database update statements and it provides specific operations to update fields. The `Insert_Statement` extends the `Update_Statement` type and is intended for database insertion.
- The `Delete_Statement` type is intended to be used to remove elements from the database.

The database statements are created by using the database session and by providing the SQL or the named query to be used.

5.1 Query Parameters

Query parameters are represented by the Parameter type which can represent almost all database types including boolean, numbers, strings, dates and blob. Parameters are put in a list represented by the Abstract_List or List types.

A parameter is added by using either the Bind_Param or the Add_Param operation. The Bind_Param operation allows to specify either the parameter name or its position. The Add_Param operation adds the parameter at end of the list and uses the last position. In most cases, it is easier to bind a parameter with a name as follows:

```
1 Query.Bind_Param ("name", "Joe");
```

and the SQL can use the following construct:

```
1 SELECT * FROM user WHERE name = :name
```

When the Add_Param is used, the parameter is not associated with any name but it as a position index. Setting a parameter is easier:

```
1 Query.Add_Param ("Joe");
```

but the SQL cannot make any reference to names and must use the ? construct:

```
SELECT * FROM user WHERE name = ?
```

5.1.1 Parameter Expander

The parameter expander is a mechanism that allows to replace or inject values in the SQL query by looking at an operation provided by the Expander interface. Such expander is useful to replace parameters that are global to a session or to an application.

5.2 Query Statements

The database query statement is represented by the `Query_Statement` type. The `Create_Statement` operation is provided on the `Session` type and it gets the SQL to execute as parameter. For example:

```
Stmt : ADO.Statements.Query_Statement := Session.Create_Statement
          ("SELECT * FROM user WHERE name = :name");
```

After the creation of the query statement, the parameters for the SQL query are provided by using either the `Bind_Param` or `Add_Param` procedures as follows:

```
Stmt.Bind_Param ("name", name);
```

Once all the parameters are defined, the query statement is executed by calling the `Execute` procedure:

```
Stmt.Execute;
```

Several operations are provided to retrieve the result. First, the `Has_Elements` function will indicate whether some database rows are available in the result. It is then possible to retrieve each row and proceed to the next one by calling the `Next` procedure. The number of rows is also returned by the `Get_Row_Count` function. A simple loop to iterate over the query result looks like:

```
while Stmt.Has_Elements loop
   Id := Stmt.Get_Identifier (1);
   ...
   Stmt.Next;
end loop;
```

5.3 Named Queries

Ada Database Objects provides a small framework which helps in using complex SQL queries in an application by using named queries. The benefit of the framework are the following:

- The SQL query result are directly mapped in Ada records,
- It is easy to change or tune an SQL query without re-building the application,
- The SQL query can be easily tuned for a given database.

The database query framework uses an XML query file:

- The XML query file defines a mapping that represents the result of SQL queries,
- The XML mapping is used by Dynamo code generator to generate an Ada record,
- The XML query file also defines a set of SQL queries, each query being identified by a unique name,
- The XML query file is read by the application to obtain the SQL query associated with a query name,
- The application uses the `List` procedure generated by Dynamo.

5.3.1 XML Query File

The XML query file uses the `query-mapping` root element. It should define at most one **`class`** mapping and several `query` definitions. The **`class`** definition should come first before any `query` definition.

```
<query-mapping>
   <class>...</class>
   <query>...</query>
</query-mapping>
```

5.3.2 SQL Result Mapping

The XML query mapping is very close to the database XML table mapping. The difference is that there is no need to specify any table name nor any SQL type. The XML query mapping is used to build an Ada record that correspond to query results. Unlike the database table mapping, the Ada record will not be tagged and its definition will expose all the record members directly.

The following XML query mapping:

```
<query-mapping>
  <class name='Samples.Model.User_Info'>
    <property name="name" type="String">
       <comment>the user name</comment>
    </property>
    <property name="email" type="String">
       <comment>the email address</comment>
    </property>
  </class>
```

```
</query-mapping>
```

will generate the following Ada record and it will instantiate the Ada container `Vectors` generic to provide a support for vectors of the record:

```
package Samples.Model is
   type User_Info is record
     Name  : Unbounded_String;
     Email : Unbounded_String;
   end record;
   package User_Info_Vectors is
      new Ada.Containers.Vectors (Index_Type   => Natural,
                                  Element_Type => User_Info,
                                  "="          => "=");
   subtype User_Info_Vector is User_Info_Vectors.Vector;
end Samples.Model;
```

A `List` operation is also generated and can be used to execute an SQL query and have the result mapped in the record.

The same query mapping can be used by different queries.

After writing or updating a query mapping, it is necessary to launch the Dynamo code generator to generate the corresponding Ada model.

5.3.3 SQL Queries

The XML query file defines a list of SQL queries that the application can use. Each query is associated with a unique name. The application will use that name to identify the SQL query to execute. For each query, the file also describes the SQL query pattern that must be used for the query execution.

```
<query-mapping>
  <query name='user-list' class='Samples.Model.User_Info'>
   <sql driver='mysql'>
     SELECT u.name, u.email FROM user AS u
   </sql>
   <sql driver='sqlite'>
      ...
   </sql>
   <sql-count driver='mysql'>
      SELECT COUNT(*) FROM user AS u
   </sql-count>
  </query>
```

```
13 </query-mapping>
```

The query contains basically two SQL patterns. The `sql` element represents the main SQL pattern. This is the SQL that is used by the `List` operation. In some cases, the result set returned by the query is limited to return only a maximum number of rows. This is often use in paginated lists.

The `sql-count` element represents an SQL query to indicate the total number of elements if the SQL query was not limited.

The `sql` and `sql-count` XML element can have an optional `driver` attribute. When defined, the attribute indicates the database driver name that is specific to the query. When empty or not defined, the SQL is not specific to a database driver.

For each query, the Dynamo code generator generates a query definition instance which can be used in the Ada code to be able to use the query. Such instance is static and readonly and serves as a reference when using the query. For the above query, the Dynamo code generator generates:

```
1 package Samples.User.Model is
2    Query_User_List : constant ADO.Queries.Query_Definition_Access;
3 private
4    ...
5 end Samples.User.Model;
```

When a new query is added, the Dynamo code generator must be launched to update the generated Ada code.

5.3.4 Using Named Queries

In order to use a named query, it is necessary to create a query context instance and initialize it. The query context holds the information about the query definition as well as the parameters to execute the query. It provides a `Set_Query` and `Set_Count_Query` operation that allows to configure the named query to be executed. It also provides all the `Bind_Param` and `Add_Param` operations to allow giving the query parameters.

```
1 with ADO.Sessions;
2 with ADO.Queries;
3 ...
4   Session : ADO.Sessions.Session := Factory.Get_Session;
5   Query   : ADO.Queries.Context;
6   Users   : Samples.User.Model.User_Info_Vector;
7   ...
8      Query.Set_Query (Samples.User.Model.Query_User_List);
```

```
9       Samples.User.Model.List (Users, Session, Query);
```

To use the `sql-count` part of the query, you will use the `Set_Count_Query` with the same query definition. You will then create a query statement from the named query context and run the query. Since the query is expected to contain exactly one row, you can use the `Get_Result_Integer` to get the first row and column result. For example:

```
1 Query.Set_Count_Query (Samples.User.Model.Query_User_List);
2 ...
3 Stmt : ADO.Statements.Query_Statement
4   := Session.Create_Statement (Query);
5 ...
6 Stmt.Execute;
7 ...
8 Count : Natural := Stmt.Get_Result_Integer;
```

You may also use the `ADO.Datasets.Get_Count` operation which simplifies these steps in:

```
1 Query.Set_Count_Query (Samples.User.Model.Query_User_List);
2 ...
3 Count : Natural := ADO.Datasets.Get_Count (Session, Query);
```

6 Model Mapping

A big benefit when using ADO is the model mapping with the Ada and SQL code generator.

The model describes the database tables, their columns and relations with each others. It is then used to generate the Ada implementation which provides operations to create, update and delete records from the database and map them in Ada transparently.

The model can be defined in:

- UML with a modeling tool that exports the model in XMI,
- XML files following the Hibernate description,
- YAML files according to the Doctrine mapping.

This chapter focuses on the YAML description.

6.1 Table definition

In YAML, the type definition follows the pattern below:

```
<table-type-name>:
  type: entity
  table: <table-name>
  description: <description>
  hasList: true|false
  indexes:
  id:
  fields:
  oneToOne:
  oneToMany:
```

The *table-type-name* represents the Ada type name with the full package specification. The code generator will add the `_Ref` prefix to the Ada type name to define the final type with reference counting. A private type is also generated with the `_Impl` prefix.

The YAML fields have the following meanings:

Field	Description
type	Must be "entity" to describe a database table
table	The name of the database table. This must be a valid SQL name
description	A comment description for the table and type definition
hasList	When **true**, a `List` operation is also generated for the type

Field	Description
indexes	Defines the indexes for the table
id	Defines the primary keys for the table
fields	Defines the simple columns for the table
oneToOne	Defines the one to one table relations
oneToMany	Defines the one to many table relations

6.2 Column mapping

Simple columns are represented within the `fields` section.

```
<table-type-name>:
  fields:
    <member-name>:
      type: <type>
      length: <length>
      description: <description>
      column: <column-name>
      not-null: true|false
      unique: true|false
      readonly: true|false
      version: false
```

The YAML fields have the following meanings:

Field	Description
type	The column type. This type maps to an Ada type and an SQL type
length	For variable length columns, this is the maximum length of the column
description	A comment description for the table and type definition
column	The database table column name
not-null	When true, indicates that the column cannot be null
unique	When true, indicates that the column must be unique in the table rows
readonly	When true, the column cannot be updated. The `Save` operation will ignore updated.

Field	Description
version	Must be "false" for simple columns

The `type` column describes the type of the column using a string that is agnostic of the Ada and SQL languages. The mapping of the type to SQL depends on the database. The `not-null` definition has an impact on the Ada type since when the column can be null, a special Ada type is required to represent that null value.

The `ADO.Nullable_X` types are all represented by the following record:

```
1    type Nullable_X is record
2       Value   : X := <default-value>;
3       Is_Null : Boolean := True;
4    end record;
```

The `Is_Null` boolean member must be checked to see if the value is null or not. The comparison operation (=) ignores the `Value` comparison when one of the record to compare has `Is_Null` set.

Type	not-null	SQL	Ada
boolean	true	TINYINT	Boolean
	false	TINYINT	ADO.Nullable_Boolean
byte	true	TINYINT	-
	false	TINYINT	-
integer	true	INTEGER	Integer
	false	INTEGER	ADO.Nullable_Integer
long	true	BIGINT	Long_Long_Integer
	false	BIGINT	ADO.Nullable_Long_Integer
identifier		BIGINT	ADO.Identifier
entity_type	true	INTEGER	ADO.Entity_Type
	false	INTEGER	ADO.Nullable_Entity_Type
string	true	VARCHAR(N)	Unbounded_String
	false	VARCHAR(N)	ADO.Nullable_String
date	true	DATE	Ada.Calendar.Time

Type	not-null	SQL	Ada
	false	DATE	ADO.Nullable_Time
time	true	DATETIME	Ada.Calendar.Time
	false	DATETIME	ADO.Nullable_Time
blob		BLOB	ADO.Blob_Ref

The `identifier` type is used to represent a foreign key mapped to a `BIGINT` in the database. It is always represented by the Ada type `ADO.Identifier` and the null value is represented by the special value `ADO.NO_IDENTIFIER`.

The `blob` type is represented by an Ada stream array held by a reference counted object. The reference can be null.

The `entity_type` type allows to uniquely identify the type of a database entity. Each database table is associated with an `entity_type` unique value. Such value is created statically when the database schema is created and populated in the database. The `entity_type` values are maintained in the `entity_type` ADO database table.

6.3 Primary keys

Primary keys are used to uniquely identify a row within a table. For the ADO framework, only the identifier and string primary types are supported.

```
<table-type-name>:
  id:
    <member-name>:
      type: {identifier|string}
      length: <length>
      description: <description>
      column: <column-name>
      not-null: true
      unique: true
      version: false
      generator:
        strategy: {none|auto|sequence}
```

The `generator` section describes how the primary key is generated.

Strategy	description
none	the primary key is managed by the application
auto	use the database auto increment support
sequence	use the ADO sequence generator

6.4 Relations

A one to many relation is described by the following YAML description:

```
<table-type-name>:
  oneToMany:
    <member-name>:
      type: <model-type>
      description: <description>
      column: <column-name>
      not-null: true|false
      readonly| true|false
```

This represents the foreign key and this YAML description is to be put in the table that holds it.

The `type` definition describes the type of object at the end of the relation. This can be the `identifier` type which means the relation will not be strongly typed and mapped to the `ADO.Identifier` type. But it can be the table type name used for another table definition. In that case, the code generator will generate a getter and setter that will use the object reference instance.

Circular dependencies are allowed within the same Ada package. That is, two tables can reference each other as long as they are defined in the same Ada package. A relation can use a reference of a type declared in another YAML description from another Ada package. In that case, `with` clauses are generated to import them.

6.5 Versions

Optimistic locking is a mechanism that allows updating the same database record from several transactions without having to take a strong row lock that would block transactions. By having a version column that is incremented after each change, it is possible to detect that the database row was modified when we want to update it. When this happens, the optimistic lock exception `ADO.Objects.LAZY_LOCK` is raised and it is the responsibility of the application to handle the failure by retrying the update.

For the optimistic locking to work, a special integer based column must be declared.

```
<table-type-name>:
  fields:
    <member-name>:
      type: <type>
      description: <description>
      column: <column-name>
      not-null: true
      unique: false
      version: true
```

The generated Ada code gives access to the version value but it does not allow its modification. The version column is incremented only by the `Save` procedure and only if at least one field of the record was modified (otherwise the `Save` has no effect). The version number starts with the value `1`. ## Objects When a database table is mapped into an Ada object, the application holds a reference to that object through the `Object_Ref` type. The `Object_Ref` tagged type is the root type of any database record reference. Reference counting is used so that the object can be stored, shared and the memory management is handled automatically. It defines generic operations to be able to:

- load the database record and map it to the Ada object,
- save the Ada object into the database either by inserting or updating it,
- delete the database record.

The Dynamo code generator will generate a specific tagged type for each database table that is mapped. These tagged type will inherit from the `Object_Ref` and will implement the required abstract operations. For each of them, the code generator will generate the `Get_X` and `Set_X` operation for each column mapped in Ada.

Before the `Object_Ref` is a reference, it does not hold the database record itself. The `ADO.Objects.Object_Record` tagged record is used for that and it defines the root type for the model representation. The type provides operations to modify a data field of the record while tracking its changes so that when the `Save` operation is called, only the data fields that have been modified are updated in the database. An application will not use nor access the `Object_Record`. The Dynamo code generator generates a private type to make sure it is only accessed through the reference.

Several predicate operations are available to help applications check the validity of an object reference:

Function	Description
Is_Null	When returning True, it indicates the reference is NULL.
Is_Loaded	When returning True, it indicates the object was loaded from the database.

Function	Description
Is_Inserted	When returning True, it indicates the object was inserted in the database.
Is_Modified	When returning True, it indicates the object was modified and must be saved.

Let's assume we have a `User_Ref` mapped record, an instance of the reference would be declared as follows:

```
1 with Samples.User.Model;
2 ...
3   User : Samples.User.Model.User_Ref;
```

After this declaration, the reference is null and the following assumption is true:

```
1 User.Is_Null and not User.Is_Loaded and not User.Is_Inserted
```

If we set a data field such as the name, an object is allocated and the reference is no longer null.

```
1 User.Set_Name ("Ada Lovelace");
```

After this statement, the following assumption is true:

```
1 not User.Is_Null and not User.Is_Loaded and not User.Is_Inserted
```

With this, it is therefore possible to identify that this object is not yet saved in the database. After calling the `Save` procedure, a primary key is allocated and the following assumption becomes true:

```
1 not User.Is_Null and not User.Is_Loaded and User.Is_Inserted
```

6.6 Loading Objects

Three operations are generated by the Dynamo code generator to help in loading a object from the database: two `Load` procedures and a `Find` procedure. The `Load` procedures are able to load an object by using its primary key. Two forms of `Load` are provided: one that raises the `ADO.Objects.NOT_FOUND` exception and another that returns an additional `Found` boolean parameter. Within the application, if the database row is expected to exist, the first form should be used. In other cases, when the application expects that the database record may not exist, the second form is easier and avoids raising and handling an exception for a common case.

```
1  User.Load (Session, 1234);
```

The `Find` procedure allows to retrieve a database record by specifying a filter. The filter object is represented by the `ADO.SQL.Query` tagged record. A simple query filter is declared as follows:

```
1 Filter : ADO.SQL.Query;
```

The filter is an SQL fragment that is inserted within the `WHERE` clause to find the object record. The filter can use parameters that are configured by using the `Bind_Param` or `Add_Param` operations. For example, to find a user from its name, the following filter could be set:

```
1 Filter.Set_Filter ("name = :name");
2 Filter.Bind_Param ("name", "Ada Lovelace");
```

Once the query filter is initialized and configured with its parameters, the `Find` procedure can be called:

```
1 Found : Boolean;
2 ...
3 User.Find (Session, Filter, Found);
```

The `Find` procedure does not raise an exception if the database record is not found. Instead, it returns a boolean status in the `Found` output parameter. The `Find` procedure will execute an SQL `SELECT` statement with a `WHERE` clause to retrieve the database record. The `Found` output parameter is set when the query returns exactly one row.

6.7 Modifying Objects

To modify an object, applications will use one of the `Set_X` operation generated for each mapped column. The ADO runtime will keep track of which data fields are modified. The `Save` procedure must be called to update the database record. When calling it, an SQL `UPDATE` statement is generated to update the modified data fields.

```
1 User.Set_Status (1);
2 User.Save (Session);
```

6.8 Deleting Objects

Deleting objects is made by using the `Delete` operation.

```
1 User.Delete (Session);
```

Sometimes you may want to delete an object without having to load it first. This is possible by delete an object without loading it. For this, set the primary key on the object and call the `Delete` operation:

```
1 User.Set_Id (42);
2 User.Delete (Session);
```

6.9 Sequence Generators

The sequence generator is responsible for creating unique ID's across all database objects.

Each table can be associated with a sequence generator. The sequence factory is shared by several sessions and the implementation is thread-safe.

The `HiLoGenerator` implements a simple High Low sequence generator by using sequences that avoid to access the database.

Example:

```
1 F  : Factory;
2 Id : Identifier;
3 ...
4   Allocate (Manager => F, Name => "user", Id => Id);
```

6.9.1 HiLo Sequence Generator

The HiLo sequence generator. This sequence generator uses a database table `sequence` to allocate blocks of identifiers for a given sequence name. The sequence table contains one row for each sequence. It keeps track of the next available sequence identifier (in the 'value column).

To allocate a sequence block, the HiLo generator obtains the next available sequence identified and updates it by adding the sequence block size. The HiLo sequence generator will allocate the identifiers until the block is full after which a new block will be allocated.

7 Database schema migration

In the lifetime of an application, the database schema may be changed and these changes must be applied to the database when a new version of the application is installed. Each application can implement its own mechanism to track the database schema change and apply the necessary change to the database when the application is upgraded.

7.1 Migration mechanism

An automatic tracking and update is too difficult for this library because such database migration is specific to each application. However, the Ada Database Objects library provides a mechanism to help in updating the database schema. The proposed mechanism is based on:

- the `ado_version` database table which tracks for each application module, a schema version. The schema version is changed by the developer when a new database schema is made for the module and some SQL migration script must be executed. The version is a simple integer that is incremented by the developer when the schema is modified.
- a set of SQL migration scripts that control the migration of a module from a version to the next one. The SQL migration scripts are written manually by the developer (or by some generation tool if they exist) when the database schema is modified.

The SQL migration scripts must be organized on the file system so that the library can find them and decide whether they must be executed or not. Each SQL migration script of a module must be stored in a directory with the name of that module. Then, each upgrade for a schema version of the module must also be stored in a directory with the name of the version. To control the order of execution of SQL migration scripts, each script file must be prefixed by a unique order. SQL scripts are executed in increasing order. Following the unique order, a tag separated by - indicates the database driver that corresponds to the SQL script. The special tag `all` indicates that the script must be executed for all database drivers. This allows to have specific SQL for different database types such as SQLite, PostgreSQL or MySQL.

To summarize, the layout is the following:

```
db/migrate/<module>/<version>/depend.conf
db/migrate/<module>/<version>/<order>-all-<name>.sql
db/migrate/<module>/<version>/<order>-<driver>-<name>.sql
```

The database migration support is decomposed in several steps each represented by a separate procedure so that it provides a fine grain control over the migration execution. For example it is possible to print the list of SQL scripts without executing them. It is also possible to perform partial migration and upgrade only a subset of the list by stopping at any time.

To proceed with the migration, the library will do the following:

- it scans the `db/migrate` directory which contains the module SQL migration scripts. For each database module, it identifies the module versions that must be upgraded. A module needs an upgrade when there is a `<version>` directory that exists and is higher than the current schema version recorded in `ado_version`. This step is done by the `ADO.Schemas.Databases.Scan_Migration` procedure. Upon completion of this step, we get a list of `Update_Type` records that indicate every upgrade that must be executed.
- the list of upgrades must then be sorted to honor the dependencies between the database modules. This step must be done to make sure that the dependent schemas are updated to the specified dependent version. This step is executed by the `ADO.Schemas.Databases.Sort_Migration` procedure.
- having a sorted list of upgrade, it is now possible to scan the directory that contains the SQL scripts (ie `db/migrate/<module>/<version>`). When reading that directory, we only look at the SQL files with the `.sql` extension. The file name must follow the pattern `<order>-<tag>-<title>.sql` (or the regular expression `[0-9]+-(all|mysql|postgresql|sqlite)-.*\.sql`). The SQL file is taken into account if it uses the tag `all` or uses the database driver name of the database being upgraded. The final list of SQL files is then sorted. This step is handled by the `ADO.Schemas.Prepare_Migration` procedure. It produces a list of files with their absolute path in the order in which they must be executed for the module upgrade for the specific version.
- the last step is to execute the SQL statements from the list of files computed previously. During this step, the SQL file is read and split into SQL statements that are executed on the database. Once every statement is executed, the next file is processed until the last one. At the end, the version associated with the database module is update to the upgraded version. If the `ado_version` table does not contain the module, an entry is created with the version. This step is made by the `ADO.Schemas.Run_Migration` procedure. The `Run_Migration` procedure will also automatically call `Prepare_Migration` to collect the list of files if necessary.

The following code extract illustrates a complete database migration support:

```
1  with ADO.Sessions;
2  with ADO.Schemas.Databases;
3  ...
4  Session : ADO.Sessions.Master_Session;
5  List    : ADO.Schemas.Databases.Upgrade_List;
6  Files   : Util.Strings.Vectors.Vector;
7  ...
8     ADO.Schemas.Databases.Scan_Migration (Session, "db/migrate", List);
9     ADO.Schemas.Databases.Sort_Migration (List);
10    for Upgrade of List loop
11       ADO.Schemas.Databases.Run_Migration
```

```
12          (Session, Upgrade, Files, ADO.Sessions.Execute'Access);
13    end loop;
```

7.2 Taking into account the migration during development

This section gives some tips to take into account database schema changes during development.

When there is a database schema change in a module, the version should be incremented and one or several SQL scripts to upgrade only to the new version are written. The schema change must be identified and analyzed either manually or with some tool. For example if a new column is added to some table, you will have to write some SQL that looks like:

```
1  ALTER TABLE awa_post ADD COLUMN `read_count` INTEGER NOT NULL;
```

When a schema change is made on the module and that module depends on another one, it is necessary for correct dependency order to write the `depend.conf` file that indicates the list of modules and their version we are depending on.

For example, if we update a module named `blogs` and that module depends on `awa` which itself depends on `ado`, the `depend.conf` file must specify these two modules with their specific versions. For example:

```
1  awa:1 ado:2
```

It is also possible to create intermediate upgrade versions for the same module. This is useful when there are incremental development steps but also when some migration becomes too complex. By spliting the complex migration in simpler and small incremental steps, this will help when the migration must be executed because it will be possible to execute one simple step at a time.

8 Troubleshooting

8.1 Change the log configuration

The ADO runtime uses the logging framework provided by Ada Utility Library. By default, logging messages are disabled and the logging framework has a negligeable impact on performance (less than 1 us per log).

You can customize the logging framework so that you activate logs according to your needs. In the full mode, the ADO runtime will report the SQL statements which are executed.

To control the logging, add or update the following definitions in a property file:

```
log4j.rootCategory=DEBUG,console,result

log4j.appender.console=Console
log4j.appender.console.level=WARN
log4j.appender.console.layout=level-message

log4j.appender.result=File
log4j.appender.result.File=test.log

# Logger configuration
log4j.logger.ADO=INFO,result
log4j.logger.ADO.Sessions=WARN
log4j.logger.ADO.Statements=DEBUG
```

The logging framework is configured by using the `Util.Log.Logging.Initialize` operation:

```
   Util.Log.Loggers.Initialize ("config.properties");
```

which can be executed from any place (but the best place is during the application start).

You can also configure the logger in Ada by using the following code:

```
with Util.Properties;
...
   Log_Config  : Util.Properties.Manager;
   ...
   Log_Config.Set ("log4j.rootCategory", "DEBUG,console");
   Log_Config.Set ("log4j.appender.console", "Console");
   Log_Config.Set ("log4j.appender.console.level", "ERROR");
   Log_Config.Set ("log4j.appender.console.layout", "level-message");
   Log_Config.Set ("log4j.logger.Util", "FATAL");
   Log_Config.Set ("log4j.logger.ADO", "ERROR");
```

```
11    Log_Config.Set ("log4j.logger.ADO.Statements", "DEBUG");
12    Util.Log.Loggers.Initialize (Log_Config);
```

The ADO runtime has several loggers, each of them can be activated separately. The following loggers are interesting:

Logger name	Description
ADO.Drivers	Database drivers and connection to servers
ADO.Sessions	Database session management
ADO.Statements	SQL statements execution
ADO.Queries	Named queries identification and retreival

8.2 Handling exceptions

Some exceptions are raised when there is a serious problem. The problem could be of different nature:

- there is a database connection issue,
- there is an SQL error,
- there is a data inconsistency.

The `ADO.Sessions.Connection_Error` exception is raised when the connection string used to access the database is incorrect. The connection string could be improperly formatted, a database driver may not be found, the database server may not be reachable.

The `ADO.Sessions.Session_Error` exception is raised when the `Session` object is used while it is not initialized or the connection was closed programatically.

The `ADO.Queries.Query_Error` exception is raised when a named query cannot be found. In that case, the SQL that corresponds to the query cannot be executed.

The `ADO.Statements.SQL_Error` exception is raised when the execution of an SQL query fails. This is an indication that the SQL statement is invalid and was rejected by the database.

The `ADO.Statements.Invalid_Column` exception is raised after the execution of an SQL query when the application tries to access the result. It is raised when the program tries to retrieve a column value that does not exist.

The `ADO.Statements.Invalid_Type` exception is also raised after the execution of an SQL query when the value of a column cannot be converted to the Ada type. It occurs if a column contains a string while the application tries to get the column as an integer or date. Similarly, if a column is null and the returned Ada type does not support the nullable concept, this exception will be raised.

The `ADO.Statements.Invalid_Statement` exception is raised when you try to use and execute a `Statement` object which is not initialized.

The object layer provided by ADO raises specific exceptions.

The `ADO.Objects.NOT_FOUND` exception is raised by the generated `Load` procedure when an object cannot be found in the database.

The `ADO.Objects.INSERT_ERROR` exception is raised by the generated `Save` procedure executed the SQL INSERT statement and its execution failed.

The `ADO.Objects.UPDATE_ERROR` exception is raised by the generated `Save` procedure executed the SQL UPDATE statement and its execution failed.

The `ADO.Objects.LAZY_LOCK` exception is raised by the generated `Save` procedure executed the SQL UPDATE statement failed and the version of the object was changed.