Introduction

Revision 2; 12 Feb 2010

By Graham Stark (graham.stark@virtual-worlds.biz)

Mill is a simple persistence system for Ada. It's based on the PHP Propel system (http://propel.phpdb.org/), which in turn is a port of the Java Torque project (http://db.apache.org/torque/). The Ada code uses the Gnade (http://gnade.sourceforge.net/) ODBC binding as its' back end. Mill is different in to Propel and Torque in that those are Object Relational Mapping systems whereas Mill is procedural, using records and functions instead of classes.

Mill is released under the GPL. It is far from complete, but it may do enough to be useful. It was written because I really liked working with Propel (I use PHP, I'm afraid..) and I was alarmed at the amount of code that seemed to be needed to get a database working in Ada (a language I otherwise have begun to really like). Mill has been developed using the Gnat compiler on a Linux box and presently is untested anywhere else. I don't see why it shouldn't work on, for example, Windows, but it uses Ada.Containers packages that, so far as I know, are implemented only by the Gnat compiler, and possibly other Gnat-specific features.

The generator is written in Python.

You need to install:

1. A recent version of Python;
2. The Cheetah Python template engine: http://www.cheetahtemplate.org/
3. The LXML library from http://codespeak.net/lxml/[1]
4. Gnade: http://gnade.sourceforge.net/
5. AUnit Ada unit testing framework (https://libre.adacore.com/aunit/main.html)[2]
6. ODBC drivers for your database and OS. So far, Mill has been tested with
    - MySQL 5.0
    - Postgres 8.x
    - IBM DB/2 version 9.1.2
    - (partially) Firebird 2.0.x

You are on your own installing these, though...

Then, just download mill.tgz and unpack it somewhere.

1 On Debian based systems (at least) you can install this via the package manager.

2 This version has been built to use version 3 of Aunit (the latest at the time of writing); you'll need to make some manual tweaks to the generated code if using version 1. Version 2 was very different and the test code would probably never work with it.

The Mill/Propel Definition files

Mill builds an SQL schema, Ada database code, and some associated files from xml definition files. These files use a slightly modified version of the Propel schema. There are two files, ***database-schema.xml***, which describes the tables and the relationships between than, and ***runtime-conf.xml***, which holds connection information and the like. Mill's xml/ directory contains quite a large example database definition, from the author's last project, and a simple example with three tables. We discuss the simple example here.

a) Database definition file database-schema.xml:

```
<?xml version="1.0"
      encoding="ISO-8859-1"
      standalone="no"
       ?>
<!DOCTYPE database SYSTEM "http://www.virtual-worlds.biz/dtds/mill.dtd">
<database name="adrs_data">

        <table name="standard_user" description="A user">

                <column name="user_id"
                        primaryKey='true'
                        type='INTEGER'
                        description=""/>
                <column name="username"
                        required="true"
                        default=''
                        type="CHAR"
                        size="16"
                        description=""/>
                <column name='password'
                        default=''
                        type='CHAR'
                        size='32' />
                <column name='salary'
                        default=''
                        type='DECIMAL'
                        size='10'
                        scale='2' />
                <column name='rate'
                        default=''
                        type='REAL'
                        />
```

```
                <column name='date_created'
                        type='DATE'
                        default='0000-00-00'
                         />
        </table>

        <table name="standard_group" description="Group a user belongs to">
                <column name="name"
                        type='CHAR'
                        size='30'
                        primaryKey='true'
                        default='SATTSIM'
                        description="Model Name"/>
                <column name="description"
                        type='CHAR'
                        size='120'
                        description="Description"/>
        </table>

        <table name="group_members" description="Group a user belongs to">
                <column name="group_name"
                        type='CHAR'
                        size='30'
                        primaryKey='true'
                        default=''
                        description="Model Name"/>
                <column name="user_id"
                        primaryKey='true'
                        type='INTEGER'
                        description=""/>
                <foreign-key foreignTable="standard_group"   onDelete="CASCADE">
                     <reference foreign="name" local="group_name"/>
                </foreign-key>
                <foreign-key foreignTable="standard_user"   onDelete="CASCADE">
                     <reference foreign="user_id" local="user_id"/>
                </foreign-key>
        </table>
</database>
```

This describes a three table database. Each table is described by a <table> element which contains <column> elements and, optionally, <foreign-key> elements. The DTD used is taken from Propel, and is unchanged except for (a) deletion of some validator elements I

didn't see the need for and (b) addition of an extra ENUM data type. Note that Mill generates SQL definitions for the tables in the order in which they appear in the xml file, so, when declaring foreign keys, the parent tables need to be declared already;

The current version supports a very limited set of datatypes, as follows:

		mapped to:			
Definition File	**Attributes**	**SQL**		**ADA**	
		type	**default**	**type**	**default**
DATE|DATETIME|TIMESTAMP	-	TIMESTAMP	db-dependent	Ada.Calendar.Time	FIRST_DATE
CHAR,VARCHAR	size	VARCHAR(size)	''	Unbounded_String	
INTEGER	-	INTEGER	0	Integer	0
REAL|DOUBLE|LONGREAL	-	REAL	0.0	Real (Mapped to Long_Float)	0.0
BOOLEAN	-	INTEGER	0	Boolean	False
DECIMAL	size,scale	DECIMAL(size,scale)	0.0	Custom Decimal types	0.0
ENUM	values	INTEGER	0	Enumerated type	first value

Obviously, this is not as complete as I'd like, but it's enough for me for now. Note:

1. Stream types (BLOB, CLOB) are not supported at all, as I can't see support for them in Gnade;
2. All string types are modelled in Ada as Unbounded_Strings and in SQL as VARCHARS. I experimented with having one Bounded_String package for each different CHAR size declaration but the generated code got increasingly messy. Instead, when saving to or querying the database, the generated Ada code simply chops these unbounded strings off at the specified length. I also experimented with using Wide_Strings in place of standard strings, but the only ODBC driver that I could get to work properly with these was DB/2. Actually, storing and retrieving multi-byte characters using the single-byte character Gnade routines does work after a fashion, provided the database is set to support it, but string lengths and the like are likely to be reported wrongly.
3. All fields are given defaults in both the generated SQL and Ada code, even if you don't specify one;
4. ENUMs are mapped to integers in SQL. The values for the enum should be entered as a space-separated list in the values attribute;
5. At present, all time types (DATETIME, TIME, DATE, TIMESTAMP) are mapped on to the TIMESTAMP type in SQL and Ada.Calendar.Time in ADA. The package db_commons has functions to map from one to the other. Note that if you supply a default for this, it is inserted verbatim in the generated SQL, but TIMESTAMP

formats differ between database implementations. Duration types are not supported at present. The default SQL timestamp strings are:

Database	Value
Postgres	TIMESTAMP '1901-01-01 00:00:00.000000'
MySQL	TIMESTAMP '0000-00-00 00:00:00.000000'
DB2	1901-01-01 00:00:00.000000
Firebird	TIMESTAMP '1901-01-01 00:00:00

The Ada Ada.Calendar.Time constant FIRST_DATE (in base_types.ads) is set to match the value for the corresponding database;

6. Primary key fields are initialised to special 'missing value' fields defined in the base_types.ads package. This is so we can more easily define a Null record in Ada for cases where a retrieval finds nothing.

b) Data source definition runtime-conf.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <config>
        <propel>
              <datasources>
                    <!-- NB ONLY 1st one gets processed -->
                    <datasource id="simple_pg">
                          <adapter>postgres</adapter>
                          <connection>
                                <hostspec>localhost</hostspec>
                                <database>simple_pg</database>
                                <username>postgres</username>
                                <password></password>
                          </connection>
                    </datasource>
                    <datasource id="simple_db2">
                          <adapter>db2</adapter>
                          <connection>
                                <hostspec>localhost</hostspec>
                                <database>simple_db2</database>
                                <username>db2inst1</username>
                                <password>xxx</password>
                          </connection>
                    </datasource>
                    <datasource id="simple_mysql">
                          <adapter>mysql</adapter>
                          <connection>
                                <hostspec>localhost</hostspec>
                                <database>simple_mysql</database>
```

```
                                        <username>root</username>
                                        <password>xxx</password>
                                    </connection>
                                </datasource>
                            </datasources>
                        </propel>
                    </config>
```

This describes connection information for the database. The format is unchanged from Propel. Some notes:

1. Presently, this file must be called runtime-conf.xml
2. multiple datasources can be defined, but only the first one is used;
3. the id field is ignored at present;
4. the database name is the name of the database as defined in ODBC, not necessarily the name the database was originally created with.

Using Mill

1. Install mill somewhere, as above;
2. Make a directory for the created files;
3. In that directory, make a directory xml. Copy a sample database and datasource file from the mill xml/simple or xml/complicated directories into that directory. The filenames need to stay the same. Edit these files to match your odbc setup and proposed database;
4. execute <mill directory>/scripts/mill.py <path to the directory you created>

If all goes well, the output directory should contain the following directories:

```
bin  <- compilation target
database <- generated sql code
etc      <- sample gnat makefile, sample odbc.ini code
src     <- the generated ADA code
tests   <- a simple test harness using AUNIT
xml     <- the directory you created above.
```

DATABASE CODE

For our simple example, for a Postgres database, Mill generates the following SQL code:

```
--
-- created on 02-01-2008 by Mill
```

```
--
drop database if exists simple_pg;
create database simple_pg with encoding 'UTF-8';
\c simple_pg;
CREATE TABLE standard_user(
       user_id INTEGER not null default 0,
       username VARCHAR(16) not null default '',
       password VARCHAR(32) default '',
       salary DECIMAL(10, 2) default 0.0,
       rate DOUBLE PRECISION default 0.0,
       date_created TIMESTAMP default TIMESTAMP '1901-01-01 00:00:00.000000',
       PRIMARY KEY( user_id )
);

CREATE TABLE standard_group(
       name VARCHAR(30) not null default 'SATTSIM',
       description VARCHAR(120) default '',
       PRIMARY KEY( name )
);

CREATE TABLE group_members(
       group_name VARCHAR(30) not null default '',
       user_id INTEGER not null default 0,
       PRIMARY KEY( group_name, user_id ),
       CONSTRAINT group_members_FK_0 FOREIGN KEY( group_name) references
             standard_group( name ) on delete CASCADE,
       CONSTRAINT group_members_FK_1 FOREIGN KEY( user_id) references
             standard_user( user_id ) on delete CASCADE
);
```

Note that:

1. Mill can handle a certain amount of SQL dialect; the DB/2, MySQL, and Firebird SQL code is a little different in preamble statements, default declarations and the syntax of primary and foreign key declarations;
2. as discussed above UTF support is currently jammed on even though Ada-side support is limited;
3. Creating a database from the schema and registering it with ODBC is not covered here. However, the etc/ directory contains a simple Unix odbc.ini file which might be of some use for this.

ADA CODE

The src/ directory contains all the generated Ada code (except the test cases). Our example simple_pg database generates the following:

base_types.adb, base_types.ads : definitions of data types and constants;

db_commons.adb, db_commons.ads : some library routines, for example conversion routines for times;

db_commons-odbc.adb, db_commons-odbc.ads: ODBC-specific library routines;

environment.adb, environment.ads: password, user and database definitions;

logger.adb, logger.ads: a really, really crude logger package;

simple_pg_data.adb, simple_pg_data.ads: this contains the Ada record definitions for our schema;

standard_group_io.adb, standard_group_io.ads: database save/retrieve/delete routines for the Group table;

group_members_io.adb, group_members_io.ads: likewise for the group_members table;

standard_user_io.adb, standard_user_io.ads: likewise for the user table.

Base_Types.ads contains some standard definitions for missing values, as discussed above, and also defintions for any needed enumerated types and fixed-point types, plus a few convenience methods.

Db_Commons contains some conversion routines, and definitions for *Criteria:* a simple way, again borrowed from Propel, for building queries in a type-safe way (see below).

Simple_Pg_Data contains data definitions for our Simple_Pg database. It is designed to contain no references to SQL or databases at all. Each table is modelled as a simple record, using the mapping and defaults discussed above. Here is the generated code for our 3-tables (comments removed):

```
with Ada.Containers.Vectors;
with Ada.Calendar;
with base_types; use base_types;
with Ada.Strings.Unbounded;
package Simple_Pg_Data is
   use Ada.Strings.Unbounded;
      type Group_Members is record
         Group_Name : Unbounded_String := MISSING_W_KEY;
         User_Id : integer := MISSING_I_KEY;
      end record;

      package Group_Members_List is new Ada.Containers.Vectors
         (Element_Type => Group_Members,
         Index_Type => Positive );

      Null_Group_Members : constant Group_Members := (
         Group_Name => MISSING_W_KEY,
```

```
            User_Id => MISSING_I_KEY
         );
         function To_String( rec : Group_Members ) return String;

         type Standard_Group is record
            Name : Unbounded_String := MISSING_W_KEY;
            Description : Unbounded_String := Ada.Strings.Unbounded.Null_Unbounded_String;
            Group_Members : Group_Members_List.Vector;
         end record;

         package Standard_Group_List is new Ada.Containers.Vectors
            (Element_Type => Standard_Group,
            Index_Type => Positive );
         Null_Standard_Group : constant Standard_Group := (
            Name => MISSING_W_KEY,
            Description => Ada.Strings.Unbounded.Null_Unbounded_String,
            Group_Members => Group_Members_List.Empty_Vector
         );
         function To_String( rec : Standard_Group ) return String;

         type Standard_User is record
            User_Id : integer := MISSING_I_KEY;
            Username : Unbounded_String := Ada.Strings.Unbounded.Null_Unbounded_String;
            Password : Unbounded_String := Ada.Strings.Unbounded.Null_Unbounded_String;
            Salary : Decimal_10_2 := 0.0;
            Rate : Real := 0.0;
            Date_Created : Ada.Calendar.Time := FIRST_DATE;
            Group_Members : Group_Members_List.Vector;
         end record;
         package Standard_User_List is new Ada.Containers.Vectors
            (Element_Type => Standard_User,
            Index_Type => Positive );
         Null_User : constant User := (
            User_Id => MISSING_I_KEY,
            Username => Ada.Strings.Unbounded.Null_Unbounded_String,
            Password => Ada.Strings.Unbounded.Null_Unbounded_String,
            Salary => 0.0,
            Rate => 0.0,
            Date_Created => FIRST_DATE,
            Group_Members => Group_Members_List.Empty_Vector
         );

   end Simple_Pg_Data;
```

For each modelled table, four things are declared:

1. the record itself;
2. a constant Null record. This is returned by some failed retrieve statements and by a successful delete statement;
3. a collection of each record, using Ada.Containers.Vectors;
4. a simple to_string print statement, useful for debugging.

In the example, the group_members table is a 'join-table' containing foreign keys pointing to the user and group tables. Mill models this by inserting a vector field in the parent Standard_Group and Standard_User tables, each containing a list of Group_Member fields.

Standard_Group_IO (and Standard_User_IO, Group_Members_IO). Each table defined in the schema is given it's own package for interacting with the database. Here is the definition file for Standard_User_IO:

```
--
-- Created by ada_generator.py on 2008-01-02 16:42:08.780684
--
with Simple_Pg_Data;
with db_commons;
with base_types;
with ADA.Calendar;
with Ada.Strings.Unbounded;


package Standard_User_IO is

   package d renames db_commons;
   use base_types;
   use Ada.Strings.Unbounded;

   function Next_Free_User_Id return integer;


   --
   -- returns true if the primary key parts of User match the defaults in
   -- Simple_Pg_Data.Null_User
   --
   function Is_Null( Standard_User : Simple_Pg_Data.Standard_User ) return Boolean;


   --
   -- Returns the single User matching the primary key fields,
   -- or the Simple_Pg_Data.Null_Standard_User record
   -- if no such record exists
   --
```

```
   function Retrieve_By_PK( User_Id : integer ) return Simple_Pg_Data.Standard_User;

   --
   -- Retrieves a list of Simple_Pg_Data.Standard_User matching
   -- the criteria, or throws an exception
   --
   function Retrieve( c : d.Criteria ) return Simple_Pg_Data.Standard_User_List.Vector;

   --
   -- Retrieves a list of Simple_Pg_Data.Standard_User retrived by
   -- the sql string, or throws an exception
   --
   function Retrieve( sqlstr : String ) return Simple_Pg_Data.Standard_User_List.Vector;

   --
   -- Save the given record, overwriting if it exists and overwrite is true,
   -- otherwise throws DB_Exception exception.
   --
   procedure Save( User : Simple_Pg_Data.User; overwrite : Boolean := True );

   --
   -- Delete the given record. Throws DB_Exception exception. Sets value to
   -- Simple_Pg_Data.Null_User
   --
   procedure Delete( Standard_User : in out Simple_Pg_Data.Standard_User );
   --
   -- delete the records indentified by the criteria
   --
   procedure Delete( c : d.Criteria );
   --
   -- delete all the records identified by the where SQL clause
   --
   procedure Delete( where_Clause : String );
   --
   -- functions to retrieve records from tables with foreign keys
   -- referencing the table modelled by this package
   --
   function Retrieve_Associated_Group_Members( Standard_User : Simple_Pg_Data.Standard_User )
return Simple_Pg_Data.Group_Members_List.Vector;

   --
   -- functions to add something to a criteria
   --
   procedure Add_User_Id( c : in out d.Criteria; User_Id : integer; op : d.operation_type:= d.eq;
```

```
join : d.join_type := d.join_and );

   procedure Add_Username( c : in out d.Criteria; Username : Unbounded_String; op :
d.operation_type:= d.eq; join : d.join_type := d.join_and );

   procedure Add_Username( c : in out d.Criteria; Username : String; op : d.operation_type:=
d.eq; join : d.join_type := d.join_and );

   procedure Add_Password( c : in out d.Criteria; Password : Unbounded_String; op :
d.operation_type:= d.eq; join : d.join_type := d.join_and );

   procedure Add_Password( c : in out d.Criteria; Password : String; op : d.operation_type:=
d.eq; join : d.join_type := d.join_and );

   procedure Add_Salary( c : in out d.Criteria; Salary : Decimal_10_2; op : d.operation_type:=
d.eq; join : d.join_type := d.join_and );

   procedure Add_Rate( c : in out d.Criteria; Rate : Real; op : d.operation_type:= d.eq; join :
d.join_type := d.join_and );

   procedure Add_Date_Created( c : in out d.Criteria; Date_Created : Ada.Calendar.Time; op :
d.operation_type:= d.eq; join : d.join_type := d.join_and );

   --

   -- functions to add an ordering to a criteria

   --

   procedure Add_User_Id_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );

   procedure Add_Username_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );

   procedure Add_Password_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );

   procedure Add_Salary_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );

   procedure Add_Rate_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );

   procedure Add_Date_Created_To_Orderings( c : in out d.Criteria; direction : d.Asc_Or_Desc );


end User_IO;
```

With luck, the comments in the above should describe what the code does adequately. Note that, for tables with children, the Vector fields are not automatically populated; the Retrieve_Associated_Group_Members function can be used for this. The procedures declared at the end are for generating a *Criteria:* a type-safe way of generating a query for creation or deletion of records borrowed from Propel. For example, suppose you want to find all users with a user_id of more than 23 and a salary of less than 25,000. You could do this:

```
...
use db_commons;
use Simple_Pg_Data;


      c : Criteria;
      l : Standard_User_List.Vector;
   begin
      Add_User_Id( c, 24, ge );
      Add_Salary( c, 25_000.0, lt );
      l := retrieve( c );
```

TEST SUITE

The test/ directory contains a simple AUnit test suite. For each record, the suite attempts to add records, delete some of them and then retrieve the remainder. As such, the suite is a good place to look for some usage examples. Note that, presently, the tests insert records containing junk data and make no checks for consistency: as such, any database with foreign key constraints will almost certainly produce failures until you manually edit the code. The test suite is built for version 3 of the AUnit Unit test package, downloadable from Adacore.

OTHER FILES

The etc/ directory contains a Unix odbc.ini file generated from the runtime-conf.xml file, and a basic Gnat project file, with the test suite as the build target. As discussed above, these files have been tested only on my machines, and almost certainly would need modification to anyone else's system, but are included here in the hope that they may be useful.