Note

MODULEWRITER: a program for automatic generation of database interfaces

Christina L. Zheng *, Fariba Fana, Poornaprajna V. Udupi, Michael Gribskov

San Diego Supercomputer Center, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0537, USA

Received 16 January 2003; received in revised form 12 February 2003; accepted 24 February 2003

Abstract

MODULEWRITER is a PERL object relational mapping (ORM) tool that automatically generates database specific application programming interfaces (APIs) for SQL databases. The APIs consist of a package of modules providing access to each table row and column. Methods for retrieving, updating and saving entries are provided, as well as other generally useful methods (such as retrieval of the highest numbered entry in a table). MODULEWRITER provides for the inclusion of user-written code, which can be preserved across multiple runs of the MODULEWRITER program.

© 2003 Elsevier Science Ltd. All rights reserved.

Keywords: Relational database; SQL; PERL; API; Object relational mapping

1. Introduction

Object relational mapping (ORM) integrates object oriented programming languages with relational databases such as ORACLE, SYBASE and MYSQL and its popularity has grown tremendously as evidenced by the growing number of ORM development projects (Bourret et al., 2000; Bretl et al., 1998; Duc and Hofer, 2002; DataBind, Hibernate, Jaxor, Jepson et al., 2000; Jgrinder, Lambert, 1998; MIDAR). ORM’s appeal originates from its ability to improve developer productivity, code maintainability, and data flexibility by isolating developers from the details of the database. The basic idea behind all ORM tools is to combine object and relational modeling approaches and to seamlessly travel between the two systems (Banerjee et al., 1987; Fussell, 1997; Keller, 1997). Extending from this basic notion current ORM tools are capable of more intricate demands such as automatically generating code for mapped objects, tackling complex database queries, and preserving key identities so as not to acquire more than one object for a row in a table. With JAVA being the most popular object oriented programming language, most ORM tools are written with that language in mind. However, this then renders a gap for developers of other object oriented programming languages such as PERL. This gap is particularly important in bioinformatics where PERL is a popular language.

The development of TANGRAM (Jepson et al., 2000), a PERL ORM, helped to fill this void. TANGRAM is a complete system for the creation, maintenance and accession of tables in a database. TANGRAM provides generic methods for database access, however, the user is still responsible for coding a database specific module in addition to providing a database schema in order to implement the TANGRAM provided methods. As many bioinformatics projects are initiated by laboratory biologists with minimal help from professionally trained programmers, TANGRAM itself does not offer a complete enough package for biologists. Here we present MODULEWRITER, a PERL ORM that automatically generates a database-specific extendable PERL application programming interface (API) module, therefore, relieving biologists of the specific burden.

The intent of the MODULEWRITER program is to provide a basic API for a database. This API is written in a high-level language (PERL) and permits a relatively
unskilled programmer to generate programs to load and retrieve data from the database without any knowledge of SQL. For this specific reason, MODULEWRITER is aimed at achieving overall simplicity not performance. MODULEWRITER’s mapping approach consists of a simple one class one table mapping. As noted by Keller (1997), the issues of maintaining a data class are in direct conflict with the issues of achieving optimum performance, therefore, MODULEWRITER’s simplistic nature does not permit us to address many of the performance issues addressed by many other ORMs such as inheritance, data persistence, and speed. Each of these issues, however, can be directly addressed and personalized by individual users with additions or modifications to the basic API provided by MODULEWRITER. MODULEWRITER’s strength lies within its simplicity, flexibility, and extensibility.

1.1. Implementation

The MODULEWRITER system comprises of the ORM program, MODULEWRITER, and the GENERIC package, which provides function templates and functions needed for database connectivity. The system produces a database specific basic SQL API with methods to access each row and each column of the tables in a database. Standard methods are created to retrieve and set values for columns in each table; to add and update rows; and a method to retrieve the last ID entered in a table. The database specific module is then placed into a directory with the same name as the database. MODULEWRITER is typically invoked from the command line:

```
MODULEWRITER -d database -h host.sdsc.edu:1234 -u user -p password.
```

Mandatory arguments:
- `d` database name,
- `h` host name where database is located,
- `u` username for the database account,
- `p` user password for the database account.

Optional arguments:
- `l` location of the new database API directory generated by MODULEWRITER,
- `c` name of existing database API with user code to be preserved in the new API (code preservation will be discussed further below).

MODULEWRITER first determines the name and target directory for the new API module based on the command line arguments. If the specified directory already exists, a unique directory is created by appending the current date to the specified directory name before creating the new database API directory. MODULEWRITER then goes on to make a database connection with the information specified in the command line using the functions found in the GENERIC package. Once a database handle has been obtained, MODULEWRITER then inquires if the user would like the API packages to have the same user name and password as the default connection parameters. These connection parameters can be changed at anytime with the connect_to_db() function but for convenience a default is set.

MODULEWRITER uses standard SQL commands to automatically identify the tables and columns in the database. Database connection is handled through the PERL DBI.PM package. MODULEWRITER should, therefore, work with any type, size, or design of SQL database, as long as it is supported by DBI.PM. We have used MODULEWRITER extensively with the MYSQL database (Dubois, 2000).

The following is an example of the application of MODULEWRITER to a gene splicing database named ‘splice’. A partial database schema for this database is shown in Fig. 1. MODULEWRITER is run with the command:

```
MODULEWRITER -d splice -h host.sdsc.edu:1234 -u user
```

A directory named ‘splice’ will be made in the current directory since an alternative directory was not specified with the `-l` option. Within the splice directory MODULEWRITER creates a database specific module with the same name as the given database name, in this case splice.pm. splice.pm contains a function to open a database connection (connect_to_db()) and a package for each table in the database. Below is a description of each of the methods in each of the packages along with examples on how to implement them in application codes. Fig. 2 shows an example script using the splice database module to manipulate and access information from the splice database.

2. Methods

2.1. connect_to_db()

connect_to_db() opens a database handle accessible by all packages within the module. MODULEWRITER stores the username, password and database host information that were given on the command line in the database module, therefore, connect_to_db() can use them as default parameters to make a database connection. These default parameters can easily be overridden by passing a hash of any or all of the parameters above. For example to use a different username and a different password from the default values, enter:

```
splice::connect_to_db({USER => $user, PASS => $pass, NAME => $name, 
```

...continued...
Fig. 1. Partial database schema for a gene splicing database named ‘splice’.

Fig. 2. Sample script using the MODULEWRITER API to access the “splice” database.

```perl
#!/usr/local/bin/perl
use strict;
use splice; # importing the database specific module created by ModuleWriter

my $new_user = "user";
my $new_pass = "pass";

# connecting to the database
splice:connect_to_db(USER=>$new_user, PASS=>$new_pass);

# connect_to_db can either be called with or without parameters

# getting a new object to access the Isoform table
my $Isoform_object = isoform->new();

# filling the values for the Isoform table
my $id = "1";
my $dna = "caaaagcaagagagagagccccccccc......";
my $protein = "MGSGVGA.......";
my $weight = "46090.39";

$ Isoform_object->Isoform_id($id);
$ Isoform_object->dna_seq($dna);
$ Isoform_object->protein_seq($protein);
$ Isoform_object->p_mol_weight($weight);
$ Isoform_object->time('NOW()');

# saving the values into a new row in the Isoform table
$ Isoform_object->save();

# retrieving information from the Isoform table
my $key = " Isoform_id";
my $value = "1";
my $@retrieve_from_isoform->retrieve($key, $value);

# obtaining the retrieved values
my $retrieved_id = $@retrieve_from_isoform[0]->Isoform_id(); # $retrieved_id = $id
my $retrieved_dna = $@retrieve_from_isoform[0]->dna_seq(); # $retrieved_dna = $dna
my $retrieved_protein = $@retrieve_from_isoform(0)->protein_seq(); # $retrieved_protein = $protein

# updating information already in the Isoform table
my $new_weight = "895.78";
$@retrieve_from_isoform[0]->p_mol_weight($new_weight);
$@retrieve_from_isoform[0]->update();

# obtaining the last id in a table
my $last_id = $Isoform_object->last_id("Isoform_id");
```
To access a different database with the same structure, such as an older version, pass a hash with the alternative database: `splice:connect_to_db({DSN => $new_database})`. MODULEWRITER stores username and password information in the database specific module creating a possible security risk. For publicly accessible systems it is important that this module be located in a secure location. It is also highly desirable that the default database account have limited privileges in the event that this information is inadvertently exposed.

### 2.2. new()

Each package contains a `new()` method to create new objects of the table (a constructor). This constructor does not take any parameters. For instance, to obtain a new object of the Isoform table named, `$isoform_object`, use `$isoform_object = isoform- > new()`. 

### 2.3. column_specific_methods()

Methods to access each of the individual columns in each table are provided. Note that column-specific methods for each package are directly determined by the individual columns in a table. For instance, the Isoform package (Fig. 1) will contain the following column specific methods: `isoform_id()`, `dna_seq()`, `protein_seq()` and `p_mol_weight()`. The time column is expected to be an automatically generated timestamp and an access method is, therefore, not provided. To access a specific column such as `isoform_id`, call the method without a parameter: `$id = $isoform_object- > isoform_id()`. To define a value for a specific column, such as `isoform_id`, call the method with the value: `$isoform_object- > isoform_id($new_id)`.

### 2.4. save()

The `save()` method saves the information in a table object by creating a new row in the relevant table. `save()` takes no parameters. To create a new row in the Isoform table filled with the values in `$isoform_object`, use `$isoform_object- > save()`.

### 2.5. update()

The `update()` method uses the information in a table object to update an existing row in a table. `update()` takes no parameters. To update an existing row in our example Isoform table, use `$isoform_object- > update()`.

### 2.6. show()

The `show()` method is used to view all values of a specific object. `show()` takes no parameters. The show method prints a description of the column values to standard output. To display the values in the Isoform table, use `$isoform_object- > show()`.

### 2.7. retrieve()

The `retrieve()` method is used to obtain all the individual column values for a specific row of a table, and to populate the fields of the corresponding table object. `retrieve()` takes two parameters for its query: a column name and its corresponding value. Calling `retrieve()` returns as many objects as there are rows matching the query, therefore, the results of this method should be stored in an array. For example, to query the Isoform table use the following syntax: `@results = $isoform_object- > retrieve(‘p_mol_weight’, ‘156.5’)`. Internally, the sql Query “SELECT * FROM isoform WHERE p_mol_weight = 156.5” is executed to retrieve this information.

### 2.8. last_id()

The `last_id()` method is used to obtain the last id in a given table. `last_id()` takes one parameter (the id field name for that table) and returns the highest numbered value in the column specified by the parameter. It is assumed that this parameter is a unique ID and that the value returned can be incremented to generate the next unique ID in a series. To obtain the last id entered into the Isoform table one would use the following: `$id = $isoform_object- > last_id(‘isoform_id’)`.

### 3. User extensions

MODULEWRITER automatically generates an API with basic functional methods for accessing a database. These basic methods can then be further expanded and/or new methods can be incorporated into this working skeleton code. This allows the user to take full advantage of all SQL functionality, for instance, by using joins to more efficiently retrieve information from multiple tables, as well as to implement functionality that may not be offered by a particular DBMS such as consistency of fields between tables or generating reports. Once the custom code is incorporated, MODULEWRITER can propagate this user code to new versions of the database API when an update is needed. As described above, the –c command line option followed by the name of the existing database API (–c file_name) is used to preserve user code from the previous module to the next. When using this option, code located between a ‘#< PROTECT@BEGIN >’ and a ‘#/PROTECT@BEGIN’ will be copied from the earlier version of the package to the new version, and placed at the beginning of the package code. These tags must be placed alone on the line above the code to be preserved and below the
4. Results and discussion

As can be seen in the example in Fig. 2, MODULEWRITER provides a simple approach to generating an immediately usable interface to an existing database. This automatically generated code can then be used as a stand-alone package or as the basis for extension. In either event, the automatically generated code greatly simplifies the developments of codes (such as CGI scripts) that access the database. In conclusion, MODULEWRITER is a PERL ORM tool which generates extendable PERL APIs to SQL databases. MODULEWRITER was developed for MYSQL databases, but should also be compatible with any SQL database supported by DBI.pm.

Acknowledgements

This work was supported by the US National Institutes of Health (CA-88351) and National Science Foundation (DBI-9975808).

References


