2010 Fifth International Conference on Software Engineering Advances

CRUD-DOM

A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms

Oscar M Pereira[1], Rui L Aguiar[2]
Instituto de Telecomunicações
University of Aveiro
Aveiro, Portugal
{omp[1],ruilaa[2]}@ua.pt

Maribel Yasmina Santos
Algoritmi Research Centre
University of Minho
Guimarães, Portugal
maribel@dsi.uminho.pt

***Abstract*—Object-oriented programming is the most successful programming paradigm. Relational database management systems are the most successful data storage components. Despite their individual successes and their desirable tight binding, they rely on different points of view about data entailing difficulties on their integration. Some solutions have been proposed to overcome these difficulties, such as Embedded SQL, object/relational mappings (O/RM), language extensions and even Call Level Interfaces (CLI), as JDBC and ADO.NET. In this paper we present a new model aimed at integrating object-oriented languages and relational databases, named CRUD Data Object Model (CRUD-DOM). CRUD-DOM relies on CLI (JDBC) and aims not only at exploring CLI advantages as preserving its performance and SQL expressiveness but also on providing a typestate approach for the implementation of the ResultSet interface. The model design aims to facilitate the development of automatic code generation tools. We also present such a tool, called CRUD Manager (CRUD-M), which provides automatic code generation with a complementary support for software maintenance. This paper shows that CRUD-DOM is an effective model to address the aforementioned objectives.**

Keywords - CRUDDO; CRUD-DOM; database; impedance mismatch.

I. INTRODUCTION

In spite of their individual successes object-oriented and relational paradigms are simply too different to bridge seamlessly leading to difficulties informally known as *impedance mismatch* [1]. The diverse foundations of the object-oriented and the relational paradigms are a major hindrance for their integration, being an open challenge for more than 45 years [2], due to the multiplicity of aspects that need to be bridged across both paradigms: imperative languages versus declarative languages; compilation and execution performance versus search performance; classes, algorithms and data structures versus relations and indexes; transactions versus *threads*; null pointers versus null for the absence of value [2], and finally, inheritance versus specialization. The impedance mismatch thus presents several challenges for developers of common applications, where often both paradigms are found. These challenges are especially noticeable in environments where production code is under strict development deadlines, and where (timely) code development efficiency is a major concern. In order to cope with the impedance mismatch issue several solutions have emerged such as language extensions (SQLJ [3], LINQ [4]), call level interfaces [5] (JDBC [6], ODBC [7] ADO.NET [8]), object/relational mappings (O/RM) (Hibernate [9], TopLink [10], LINQ [4]) and persistence frameworks (JDO [11], JPA [12]). Language extensions may provide static syntax and type checking but always rely on proprietary standards. Call level interfaces, despite their performance, provide no static syntax or static checking. O/RM have the advantage of treating data as objects but do not take the advantage of the database engine performance and further rely on proprietary standards. Persistent frameworks have the same drawbacks as O/RM.

Despite CLI drawbacks, they cannot be discarded as an important and valid option whenever performance and SQL expressiveness are considered key issues [2]. CLI provide mechanisms to encode Create, Read, Update and Delete (CRUD) expressions inside strings, easily incorporating the power and the expressiveness of SQL. Thus, power and expressiveness are crucial advantages of CLI but this comes with unavoidable and important drawbacks (see detailed discussion in section III). Our work aims to overcome these drawbacks. For such, we developed a model, known as CRUD Data Object Model (CRUD-DOM) where each CRUD expression is wrapped into an object-oriented component, known as CRUD Data Object (CRUD-DO). Furthermore, we developed a tool addressing automatic CRUD-DO generation relying on user SQL statements written from scratch. This tool is known as CRUD Manager (CRUD-M).

This paper is organized as follows. Section II presents related work. Section III highlights the *impedance mismatch* problem. Section IV describes our proposed model (CRUD-DOM), while section V presents the automatic code generation tool (CRUD-M). Section VI presents performance assessment and finally, Section VII presents the final conclusion.

Throughout this paper all examples are based on Java, SQL Server 2008 and JDBC (CLI) for SQL Server (sqljdbc4). Code snippets may not execute properly since we will only show the relevant code for the points under discussion. For conciseness, Figure 1 presents a partial view of a database schema which will be used throughout the

978-0-7695-4144-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSEA.2010.25

IEEE computer society

examples of this paper. This database is associated with the academic life, as we expect to be easily understood.

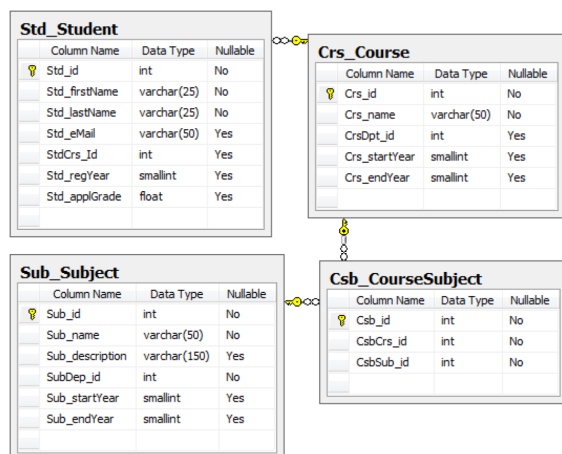

Figure 1. Partial view of the database schema

II. RELATED WORK

This section presents the different approaches for the integration of the object-oriented and the relational paradigms. As a well-known problem in industry, multiple techniques have been addressing the impedance mismatch problem.

Embedded SQL [13] is a method for writing SQL statements in-line with regular source code of the host language inside source files. These files are then pre-processed in order to check the correctness of the SQL statements namely against the database schema, host language data type and SQL data type checking, and finally syntax checking of the SQL constructions. SQLJ [3] is an example of an Embedded SQL standard which provides language extensions for embedding SQL statements in regular Java source files. Some SQLJ disadvantages, which are common to most Embedded SQL technologies: 1) SQLJ relies on an extra standard; 2) SQLJ does not decouple SQL statements from regular source code; 3) SQLJ does not provide a clean object-oriented interface to the assisted application; 4) SQLJ does not provide assistance regarding the maintenance of SQL statements; 5) SQLJ requires a JVM (Java Virtual Machine) built in the database. In practice, embedded SQL has never been widely adopted by end users.

Object-relational mapping [14-15] is a programming technique aiming at enforcing relational data models to be closely aligned with the object-oriented paradigm. The relational to object-oriented translation is driven by an explicit mapping (generally in XML) or by schema annotations (inside the source code file). Much of the enforcement is on behalf of getting an object-oriented logic access layer coping with the impedance mismatch [1] issue. Every relational concept must, somehow, have its corresponding concept(s) in the object-oriented paradigm. Very often, the translation is not straightforward leading to complex translations, as the case of the relationship and specialization concepts. In these cases, besides the aforementioned hindrance, the relational model lacks essential conceptual information obliging oneself to an extra effort on defining relationship direction, cardinality, etc. Nevertheless, O/RM techniques have been quite successful, either as commercial products (e.g., Oracle TopLink [10], ADO.NET Entity Framework [16], LINQ [4]) or as open source projects (e.g., Hibernate [17]). Albeit this achieved success, well known O/RM drawbacks are unavoidable: 1) each O/RM programming technique relies on proprietary standards introducing new mapping schemas and new SQL-equivalent manipulation languages; 2) O/RM entails an additional effort to map the relational model into the object-oriented model; 3) performance and expressiveness are the two main O/RM penalties.

Safe Query Objects [18] combine object-relational mapping with object-oriented languages to specify queries using strongly-typed objects and methods. They rely on Java Data Objects to provide strongly-typed objects and also to provide data persistence. Safe Query Objects are a promising technique to express queries but share most of the aforementioned drawbacks of O/RM namely regarding performance and SQL expressiveness.

SQL DOM [19] generates a Dynamic Link Library containing classes that are strongly-typed to a database schema. These classes are used to construct dynamic SQL statements without manipulating any strings. As Safe Query Objects, SQL DOM does not take the full advantage of SQL expressiveness and also exhibits very poor results regarding performance.

Static Checking of Dynamically Generated Queries [20] presents a solution based on static string analysis of Java programs to find out where SQL statements are being constructed. The main idea is to find out all possible combinations of distinct SQL statements and then analyze them regarding their syntax and their type mismatch errors. This approach does not affect system performance but exhibits some drawbacks as: 1) all source code is hand written from string concatenation till JDBC execution context; 2) it does not provide any object-oriented view of the SQL statement execution context.

In order to overcome the drawbacks of these techniques, we aim to explore CLI, namely through JDBC, for addressing the impedance mismatch problem.

III. IMPEDANCE MISMATCH: COMMON JDBC DRAWBACKS

JDBC is a common tool for integrating relational databases with Java (object-oriented programming language). JDBC is also a representative of the typical challenges. As such, we will explore JDBC as a target tool.

Thus, this section aims to emphasize common drawbacks regarding the utilization of JDBC including the *ResultSet* interface. The drawbacks may be split into four categories: 1) the process for editing SQL statements; 2) the process for retrieving data from returned relations; 3) the process of updating databases through *CONCUR_UPDATABLE ResultSets*; 4) protocols of *ResultSet* interface regarding its usability.

Figure 2 presents a simple example which comprises some of the drawbacks related to categories 1), 2) and 3). This example is used in the following paragraphs, which describe JDBC drawbacks:

a) There is no easy way to link CRUD expressions and their results to the application they assist. CLI provide services to ease the integration of object-oriented applications and relational databases but relevant issues are not overcome such as string concatenation (Figure 2: lines 22-24) and conversion between relational and object-oriented paradigms (Figure 2: lines 27, 28, 30).

```
20    void getCourses_( int dptId, int startYear )
21                             throws Exception {
22      sql="select * from Crs_Course"+
23              "where CrsDpt_id="+dptId+"and "+
24              "Crs_startYear="+startYear+";";
25      rs=st.executeQuery(sql);
26      while ( rs.next() ) {
27        crsId = rs.getInt("Crs_id");
28        crsName = rs.getString("Csr_name");
29        // other attributes
30        rs.updateString( "Crs_name", "John Nace");
31        // other updates
32        rs.updateRow();
33      }
34    }
```

Figure 2. Some JDBC drawbacks

b) Editing CRUD expressions and access to their results is tricky and error-prone. CRUD expressions are constructed by concatenating strings and access to their results is achieved by reading attribute by attribute in a row by row basis. Some of the most usual errors are: a) concatenation errors - missing space between lines (lines 22, 23), missing space before "and" (Figure 2: line 23); b) type mismatch error - argument *startYear* and column *Crs_startYear* (Figure 2: lines 20, 24); c) retrieving data - misspelled column name (Figure 2: line 28);

c) Errors cannot be checked for correctness at compile time, addressed in [20]. None of the previous errors can be caught at compile time demanding great accuracy while editing code in order to prevent additional time on testing, debugging and future maintenance.

d) CRUD expressions are awkward regarding their maintenance, addressed in [21]. CRUD expressions (construction and execution) comprise many different entities grouped in three classes: SQL syntax, CLI services and database schema. While SQL syntax and CLI services can be considered stable, database schema is a dynamic entity. Database schema may change for many reasons, as initial error on conceptual model or the emerging of new requirements, which usually happens several times during the development process and even also after application deployment.

e) CRUD expressions are vulnerable to SQL injection attacks, addressed in [22]. This issue is not addressed in the current version of CRUD-DOM.

f) *ResultSet* protocols, *ResultSet* interface has dozens of states, dealing with different combinations of *ResultSet* instantiation, direction, access, updates, etc. The developer is before a huge task to become aware of how to use the *ResultSet* interface. Figure 3 presents a partial view of the *ResultSet* interface. Each *ResultSet* state has its own usage protocol gathering a subgroup of the methods presented in Figure 3.

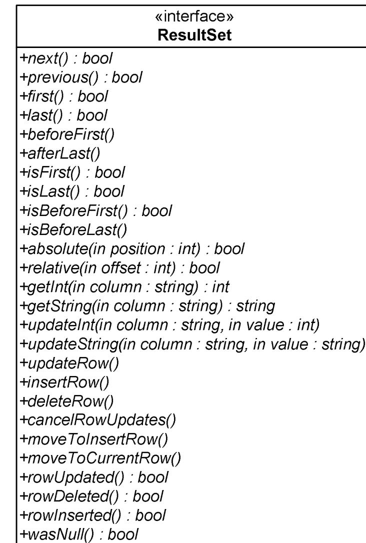

Figure 3. Partial view of the *ResultSet* interface

Some of the aforementioned drawbacks have already been individually addressed (see citations). In this paper we will present a simple, integrated and unified alternative to overcome all the aforementioned drawbacks, except the SQL injection attack.

IV. CRUD-DOM

CRUD-DOM is our abstract model aimed at coping with the drawbacks described in section III. CRUD-DOM must assure that applications and databases bridge seamlessly and also that CRUD-DOs may be automatically generated by a tool (in our case, CRUD-M).

Before we delve into the CRUD-DOM issue we will present a concise overview of CRUD expressions.

A. CRUD Expressions

CRUD expressions are the basic entities from which CRUD-DOM specification must evolve. Therefore, before proceeding with the CRUD-DOM specification, it is advisable to briefly survey CRUD expressions in order to be aware of the CLI context in which they are used.

CRUD expressions comprise the four basic SQL statements for accessing information in databases: *Select*, *Insert*, *Update* and *Delete*. While *Insert*, *Update* and *Delete* statements are used to alter the state of databases, *Select* statement allows the implementation of several views of the database. Hence, CRUD expressions may be grouped into

two categories: "query CRUD expressions" (Q-CRUD) whenever involving a *Select* statement; and "execute CRUD expressions" (E-CRUD) whenever involving an *Insert*, *Update* or *Delete* statement. The corresponding CRUD-DOs share some source code but relevant differences must be emphasized. The most relevant difference is that Q-CRUD expressions return relations from the database therefore requiring specific processing, as seen in Figure 2 (lines 26-28). Additionally, in some circumstances and also for certain Q-CRUD expressions it is possible to instantiate updatable *ResultSet*s. Updatable *ResultSet*s provide embedded protocols to update, to delete and to insert data in databases. Figure 2 (lines 30-32) concisely presents a case for the update situation. Q-CRUD expressions underlying updatable *ResultSet* are named as Active Q-CRUD expressions (AQ-CRUD). The remaining non updatable are known as Passive CRUD expressions (PQ-CRUD).

B. CRUD-DOM Details

We will present CRUD-DOM by enumerating and describing the fundamental features for each type of CRUD expression: E-CRUD, PQ-CRUD and AQ-CRUD. Afterwards, we will present class diagrams for each type of CRUD expression. For all presented examples we assume that:

- "CruddoName" is the name for all used types of CRUD expressions;
- Q-CRUD expression is "*select co1A, colB from table where colA>param*" where *colA* is *integer* and *colB* is *string*;
- E-CRUD is any *delete*, *update* or *insert* SQL statement with one parameter (*param*) of type integer.

Features:

All CRUD expressions share the following features:

- Each CRUD expression must have a unique name which will be used to build some names of CRUD-DO classes;
- Every CRUD-DO is built around one class, known as invocation class, and among other things, the class is responsible for the execution of the CRUD expression. The name of the invocation class is the same as the one given to the CRUD expression.
- The invocation class has only one constructor with one argument, the type of which is *Connection* (java.sql).
- The invocation class has one method named as *execute* which is responsible for the execution of the CRUD expression. This method returns no value and has as many arguments as the number of the CRUD expression parameters. The name, type and order of the arguments depend on the name, type and order of CRUD expression parameters.

All CRUD-DOs derived from E-CRUD expressions share the following feature:

- The invocation class has a method with the following signature: *int getAffectedRows()*; this method returns the number of affected rows by the execution of the E-CRUD expression.

Figure 4 presents the class diagram for the E-CRUD expression example, CruddoName.

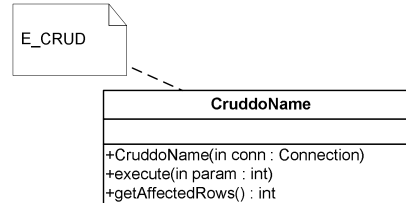

Figure 4. Invocation class for E-CRUD expressions

All CRUD-DOs derived from Q-CRUD expressions share the following feature:

- The invocation class has one method with the signature: *boolean moveNext()*; it is responsible for indicating if there is another row and for moving the cursor down one row from the current position;

Q-CRUD expressions have no concrete instances. They are super types for PQ-CRUD and AQ-CRUD expressions.

If *ResultSet* is created as scrollable, the invocation class implements other scrollable methods.

All CRUD-DOs derived from PQ-CRUD expressions share the following features:

- Extend features of Q-CRUD expressions;
- The invocation class has one method with the following signature: *CruddoName_readTuple beginRead()*;

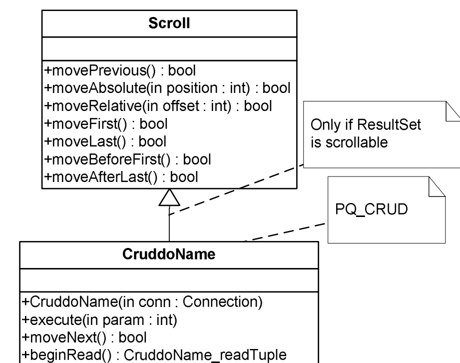

Figure 5. Invocation class for PQ-CRUD expressions

- *CruddoName_readTuple* class, known as the access class, implements one method, generally known as access method, for each attribute of the returned relation. Each access method has the following signature *JavaDataType getAttributeName()* where *JavaDataType* is the correspondent java data type for the SQL data type and the method's name is built by concatenating the name of the attribute (first letter

converted into uppercase) with the prefix *get*. Figure 5 and Figure 7 present the class diagrams for PQ-CRUD expressions.

All CRUD-DOs derived from AQ-CRUD expressions share the following features:

- Extend features of Q-CRUD expressions;
- The invocation class may provide any subset of the following four features: *readable*, *updatable*, *insertable* and *deletable*; whenever provided, the *readable* feature may also be included in the remaining features to improve their usability;
- If CRUD-DO is *readable* it implements one method with the following signature: *CruddoName_readTuple beginRead()*;
- If CRUD-DO is *updatable* it implements one method with the following signature: *CruddoName_updateTuple beginUpdate()*;
- If CRUD-DO is *insertable* it implements one method with the following signature: *CruddoName_insertTuple beginInsert()*;
- If CRUD-DO is *deletable* it implements one method with the following signature: *delete()*;
- *CruddoName_readTuple* class: previously explained for PQ-CRUD;
- *CruddoName_updateTuple* and *CruddoName_insertTuple* classes provide functionalities easily perceived from *CruddoName_readTuple* class: access methods have *set* as prefix instead of *get*;.
- The *delete* method, deletes the current row from the *ResultSet*.

Figure 6, Figure 7, Figure 8 and Figure 9 present the class diagrams for AQ-CRUD expressions.

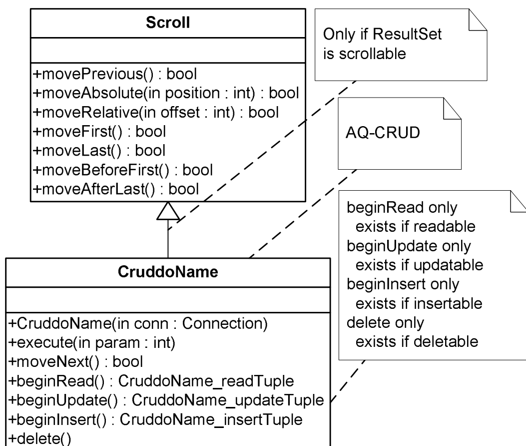

Figure 6. Invocation class for AQ-CRUD expressions

Class diagrams have been presented for each type of CRUD expression. To completely understand the class diagrams it is necessary to have an understanding of how the *ResultSet* interface is implemented. Original *ResultSet* method names have been renamed and some new ones have been included. Renamed methods are easily identified: *next->moveNext*, *previous->movePrevious*, etc. Only a subgroup of all methods has been presented in order to avoid overcrowd the class diagrams.

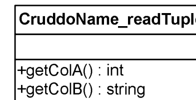

Figure 7. Readable class for Q-CRUD expressions

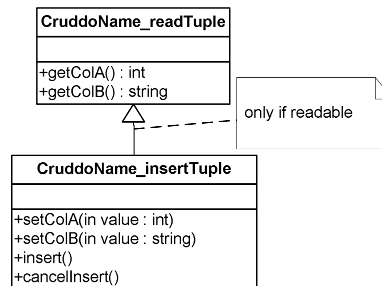

Figure 8. Insertable class for AQ-CRUD expressions

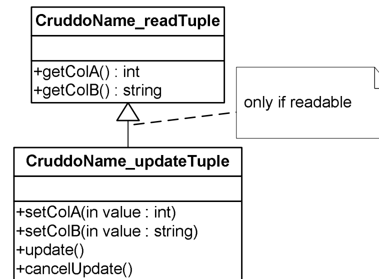

Figure 9. Updatable class for AQ-CRUD expressions

V. CRUD MANAGER

The CRUD-M addresses the features for CRUD-DO automatic code generation and also for CRUD-DO maintenance. No special programming skills should be required to use CRUD-M and learning time should be minimal. CRUD-M usage is centered in a GUI component presented in Figure 10. Figure 10 shows a concrete example for an AQ-CRUD expression, called *GetCourses*, which is readable, updatable and insertable but not deletable. Figure 11 shows the usage of CRUD-DO *GetCourses* from the application point of view. As one can see, the integration is seamless regarding impedance mismatch. Additionally, an initial approach for the implementation of *ResultSet* as a typestate [23] component is provided improving this way CRUD-DO usability. This may be verified, as an example, by the definition of the *GetCourses_readTuple* class (Figure 11, lines 44,45) which provides a coherent protocol for retrieving data from the *ResultSet*.

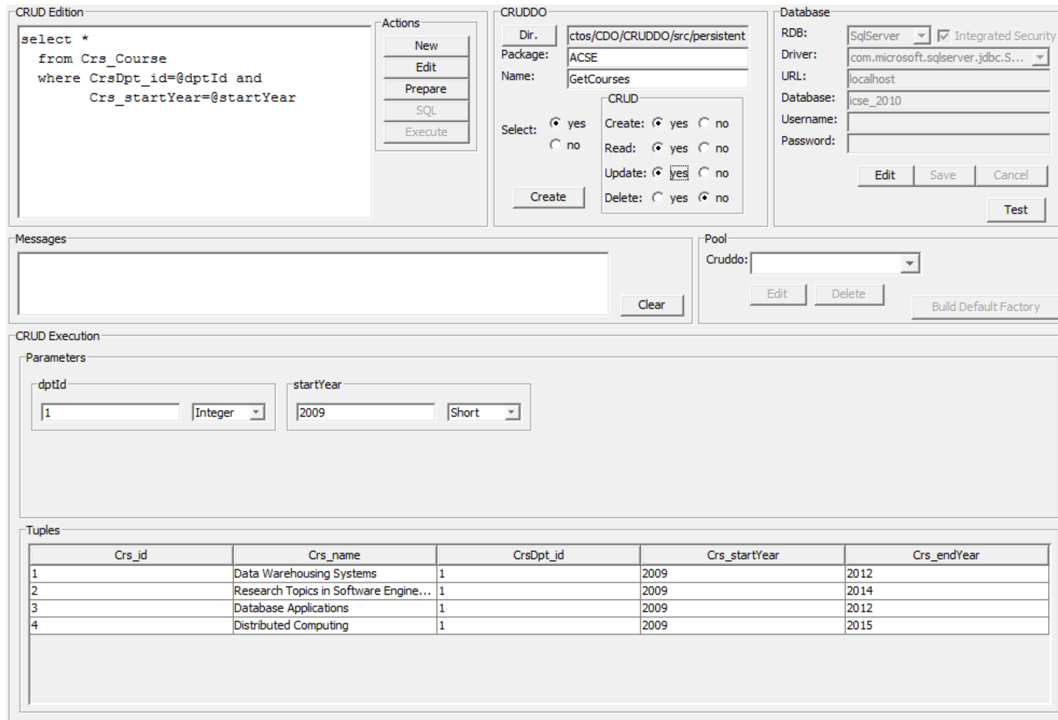

Figure 10. CRUD-M GUI

```
40   void getCourses( int dptId, short startYear )
41                throws Exception {
42     GetCourses gc=new GetCourses(connection);
43     gc.execute(dptId, startYear);
44     GetCourses_readTuple rt=gc.beginRead();
45     String name=rt.getCrs_name();
46     // code
47     GetCourses_updateTuple ut=gc.beginUpdate();
48     ut.setCrs_endYear((short)2015);
49     // code
50     ut.update();
51   }
```

Figure 11. *GetCourses* from the application point of view

The CRUD-M encompasses five main blocks as depicted in Figure 12. User launches CRUD-M and defines which database is going to be used. Then, "Schema Reader" reads the schema of the database. From now on, users may edit and/or maintain CRUD expressions. "CRUD Editor" provides a context where CRUD expressions may be edited. "CRUD Execution Unit" may help "CRUD Editor" in some specific tasks as defining SQL parameters and executing statements against the database. After executing successfully a SQL statement against the database, users are allowed to create CRUD-DO which will be accomplished by "CRUD-DO Generator". "CRUD Maintenance" parses CRUD-DO and retrieves the underlying CRUD expression to be reedited by "CRUD Editor". A more detailed description for each bock follows:

Schema Reader: this component reads the schema of the database which is mainly used to automatically suggest the Java data types for parameters of CRUD expressions.

CRUD Editor: CRUD Editor is a text editor where CRUD expressions may be written from scratch. Parameters defined in runtime must be identified through a unique name preceded by a '@' character. These names will be used for the arguments of the invocation classes. In our example we have defined two parameters: *dptId* and *startYear*.

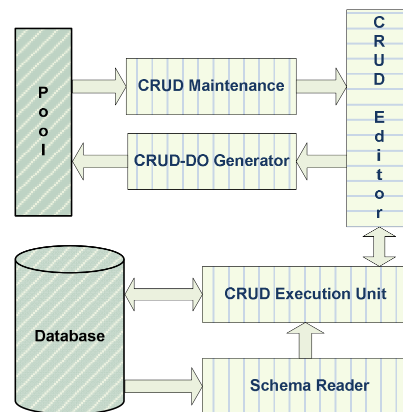

Figure 12. Block diagram of the CRUD-M

CRUD Execution Unit: CRUD Execution Unit is responsible for three tasks: 1) providing, whenever necessary, input data components for SQL parameters. Each input component is identified by the name of the associated parameter and has a

default Java Data Type derived from the database schema. Users may select another Java Data Type becoming responsible for their decision; 2) executing the edited CRUD expression against the database proving this way an expedite and integrated tool for evaluating the correctness of CRUD expressions and also for testing the outcome of CRUD expressions. Developers are relieved to write source code to test and debug their CRUD expressions; 3) formatting a table in runtime to present the content of returned relations, whenever the underlying CRUD is a Q-CRUD expression. This visualization allows developers to have an immediate visual feedback about the retuned data and easily evaluate the outcome of Q-CRUD expressions. In our example, the Q-CRUD expression returned a relation with 4 rows and 5 attributes.

CRUD-DO Generator: CRUD-DO Generator creates automatically all the necessary classes for the underlying CRUD expressions. For all types of CRUD expressions, users must input some additional information, as: CRUD-DO's name, package's name, type of CRUD expression, pool directory for CRUD-DOs, etc. Some additional information is required if CRUD expression is of type AQ-CRUD, as if it is *readable*, *insertable*, *updatable* or *deletable*.

CRUD Maintenance: this component keeps track of all existing CRUD-DOs in the pool directory. Any CRUD-DO in the pool directory may be selected for editing or to be deleted. If it is selected for editing, the underlying CRUD expression is retrieved from the invocation class and presented by the CRUD editor. From now on, the CRUD expression may be retested or reedited to update the current CRUD-DO or even to create a new one.

VI. PERFORMANCE ASSESSMENT

Performance is an indicator of how well a software system or component meets its requirements for timeliness [24]. There are two dimensions: responsiveness and scalability. This paper mostly discusses responsiveness aspects and scalability will be considered in a near future. Hereafter, performance should be understood as the responsiveness dimension.

The performance assessment here presented covers standard JDBC and CRUD-DOM solutions. All assessments share the same platform: PC - Dell Latitude E5500; CPU - Intel Duo Core P8600 @2.40GHz; RAM - 4.00 GB; OS - Windows Vista Enterprise Service Pack 2 (32bits); Java SE 6; JDBC(sqljdbc4); NetBeans 6.5.1; SQL Server 2008 running on localhost; minimum used counting interval – 0,1ms. In order to provide an ideal test environment the following actions were taken: the running thread was given the highest priority and all non essential processes/services were cancelled.

TABLE I presents the results of all measurements and also relevant supplementary information (at the bottom part of the table) to understand its contents. Depending on the *ResultSet* type and on the performed operation, 10 types of conditions were defined for an AQ-CRUD expression. AQ-CRUD expression was the chosen type because it is the most susceptible CRUD type regarding CRUD-DOM architecture in terms of wrapping classes and therefore regarding overall performance. For the 10 conditions 30 assessments were carried out for 3 contexts: JDBC, DOM and Submit. All values, for each condition, represent the time required to compute N cycles as explained in the next topics. Figure 13 and Figure 14 present a partial view of the source code to execute an update and delete operation, respectively, for each of the 3 contexts. Each context is detailed in the next paragraphs.

JDBC: this context represents, for each individual condition, the normalized performance. The value 1.000 represents, for each condition, 100.0 ms in which are computed N cycles of standard JDBC code. This value N is computed in an interactive way and will be used in the remaining 2 contexts to evaluate the time required to compute the equivalent source code. The important issue in this context is that the updated information (update, insert and delete operations) is not submitted to the database, see Figure 13 and Figure 14 (JDBC). This way, the results will only depend on the implemented approaches avoiding overheads from external components.

DOM: this context represents the normalized performance to execute each equivalent CRUD-DOM condition with the same number of N cycles, see Figure 13 and Figure 14 (CRUD-DOM). Akin to JDBC, the information is not submitted to the database.

TABLE I. ASSESSMENT FOR JDBC AND CRUD-DOM

Id	Rs	O	N	JDBC	DOM	Submit	%
0	FR	R	40,668	1,000	1,017	0	1.7
1	FU	R	35,370	1,000	1,013	0	1.3
2	FU	U	34,620	1,000	1,021	204,352	0.01
3	FU	C	39,650	1,000	1,029	207,367	0.01
4	FU	D	3,010e3	1,000	1,155	735,280e3	<0.001
5	SR	R	34,320	1,000	1,012	0	1.2
6	SU	R	34,670	1,000	1,017	0	1.7
7	SU	U	35,018	1,000	1,040	220,779	0.02
8	SU	C	38,671	1,000	1,041	215,189	0.02
9	SU	D	3,071e3	1,000	1,168	758,940e3	<0.001
Rs(ResultSet Type): F –forward only, S – scrollable, R – only readable, U – readable and updatable;							
O (Operation): R – read, U – update, C – insert, D – delete.							
Performed operation/loop: C – two strings and two integers; R – two strings and two integers; U – two strings and two integers; D – one row.							
% = [(DOM+Submit)-(JDBC+Submit)]/(JDBC+Submit)							

Submit: this context is the total time required to execute N cycles and also for submitting the information to the database for each condition, see Figure 13 and Figure 14 (Submit). We present a single column because, in this

context, standard JDBC and CRUD-DOM performances could not be distinguished. Figure 13 and Figure 14 present the code for standard JDBC (Submit).

Surprisingly, the obtained results show that performances for JDBC and CRUD-DOM contexts are very similar. if D operation is not considered, the maximum percentage difference is 4.1% and was on SU-C (Id=8). Some additional tests were carried out to understand these results. We came into the conclusion that the overhead introduced by *ResultSet* methods (shared by both approaches – *getInt*, *getString*, *updateInt*, *updateString*) consumed almost all the required time to compute JDBC and DOM contexts. This assertion has been proved after removing those operations from both contexts. Those methods cannot be avoided leading to no other option than taking them as part of the overall performance assessment.

The Delete (D) operation introduces an overhead close to 16%. This result comes from the fact that JDBC and CRUD-DOM contexts are very short of code, see Figure 14 (JDBC and CRUD-DOM). Any additional code may convey a significant overhead as is the case of *ru.delete()* in spite of being an empty method.

```
2278    // JDBC (update)
2279  while (cycle>0) {
2280    rs.updateString("std_firstName",firstName);
2281    rs.updateInt("std_id",id);
2282    rs.updateString("std_lastName",lastName);
2283    rs.updateInt("stdCrs_Id",courseId);
2284    // rs.updateRow(); disabled
2285    cycle--;
2286  }
2287  // CRUD-DOM (update)
2288  while (cycle>0) {
2289    u=ru.beginUpdate();
2290    u.setStd_firstName(firstName);
2291    u.setStd_id(id);
2292    u.setStd_lastName(lastName);
2293    u.setStdCrs_id(courseId);
2294    u.updateTuple();  // empty method
2295    cycle--;
2296  }
2297  // Submit (update)
2298  while (cycle>0) {
2299    rs.updateString("std_firstName",firstName);
2300    rs.updateInt("std_id",id);
2301    rs.updateString("std_lastName",lastName);
2302    rs.updateInt("stdCrs_Id",courseId);
2303    rs.updateRow();
2304    cycle--;
2305  }
```

Figure 13. Update operation

In many situations performances of JDBC and CRUD-DOM contexts may be considered equivalent or at least similar. But JDBC and CRUD-DOM contexts, for most of the conditions, do not express real situations. Conditions where update, delete or insert operations are carried out, the information must be submitted to the database. These conditions have been addressed in the Submit context. Submit column (TABLE I) shows that the normalized values are much higher than the ones for JDBC and CRUD-DOM contexts. Submit context tells us that the time to accomplish the submission task takes at least 200 times the ones for the JDBC and CRUD-DOM contexts in spite of the optimal running environment.

Regarding delete operations, they take at least 700,000 times more. This value results from the fact that the JDBC and CRUD-DOM contexts, as mentioned before, are very short of code conveying a very high weight to the effective *delete* operation.

As a final summary, column % shows the overall performance decay in percentage. The results were obtained from the formula shown at the bottom of the TABLE I which stresses the total performance decay for real situations. The maximum decays come from "read" operations which oscillate between a maximum of 1.7% and a minimum of 1.2%. Regarding "update" and "insert" operations performance decays oscillate between 0.01% and 0.02%. Regarding "delete" operations, performance decay is under 0.001%. From the obtained results, loosely speaking, we may argue that the overhead introduced by CRUD-DOM may be considered as irrelevant in a Submit context.

```
2306  // JDBC (delete)
2307  while ( cycle>0 ) {
2308    //rs.deleteRow();  // disabled
2309    rs.next();
2310    cycle--;
2311  }
2312  // CRUD-DOM (delete)
2313  while ( cycle>0 ) {
2314    ru.delete();  // empty method
2315    ru.moveNext();
2316    cycle--;
2317  }
2318  // Submit (delete)
2319  while ( cycle>0 ) {
2320    rs.deleteRow();
2321    rs.next();
2322    cycle--;
2323  }
```

Figure 14. Delete operation

VII. CONCLUSION

CRUD-DOM proved to be an effective model for wrapping customized CRUD expressions. The main positive advantages of this model are: 1) it encapsulates CRUD expressions and exposes an object-oriented interface to the assisted application; 2) the interface is strongly-typed; 3) it is amenable to the development addressing automatic code generation; 4) it copes with requirements as SQL expressiveness and system performance; 5) it does not rely on any complementary or proprietary technology; 6) it promotes the development of intermediate access layers decoupled from applications and databases. Regarding CRUD-DOM performance, in spite of the limited range of tests involving read and write operations, this early assessment suggests that any additional effort to improve its performance should start and be focused on "read" operations. Performance of the remaining operations in the Submit context is not sensible to the JDBC or to the CRUD-DOM approach. It is beyond the programmer's scope. Thus, it is expected that any improvement in the source code

should have a negligible impact on performance.

The automatic code development tool, CRUD-M, designed as proof of concept, proved to be an efficient tool addressing all features of CRUD-DOM in an integrated way. Programmers are only required to input customized SQL statements. CRUD-M relieves programmers from writing and testing any source code. Additionally, it provides an interactive GUI where programmers are guided step by step, since the editing of CRUD expressions till the creation of CRUD-DO.

In order to improve CRUD-DO performance, we are already addressing some key issues such as support for *PreparedStatement*s, implementing concurrency between CRUD-DOs and also implementing instance pooling for CRUD-DOs.

REFERENCES

[1] M. David, "Representing database programs as objects," in *Advances in Database Programming Languages*, ed N.Y.: ACM, 1990, pp. 377-386.

[2] W. Cook and A. Ibrahim. (2010 Mar 20). *Integrating programming languages and databases: what is the problem?* Available: http://www.odbms.org/experts.aspx#article10

[3] *Part 1: SQL Routines using the Java (TM) Programming Language*, 1999.

[4] A. Hejlsberg. (2010 Mar 15). *The LINQ Project*. Available: http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

[5] ISO. (2003, 2010) ISO/IEC 9075-3:2003. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134

[6] S. Microsystems. (2010 Feb 27). *JDBC Overview*. Available: http://java.sun.com/products/jdbc/overview.html

[7] Microsoft. (2010 Mar 18). *Microsoft Open Database Connectivity*. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx

[8] Microsoft. (2010 Mar 12). *Overview of ADO.NET*. Available: http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx

[9] Hibernate. (2010 Feb 24). *Hibernate*. Available: http://www.hibernate.org/

[10] Oracle. (2010 Mar 15). *Oracle TopLink*. Available: http://www.oracle.com/technology/products/ias/toplink/index.html

[11] S. Microsystems. (2010 Mar 23). *Java Data Objects (JDO)*. Available: http://java.sun.com/jdo/

[12] Sun.Microsystems. (2010 Feb 25). *JPA - Java Persistent API*. Available: http://java.sun.com/javaee/technologies/persistence.jsp

[13] J. W. Moore, "The ANSI binding of SQL to ADA," *Ada Letters,* vol. XI, pp. 47-61, 1991.

[14] W. Keller, "Mapping Objects to Tables - A Pattern Language," in *European Conference on Pattern Languages of Programming Conference (EuroPLoP)*, Irsse, Germany, 1997.

[15] R. Lammel and E. Meijer, "Mappings Make data Processing Go 'Round: An Inter-paradigmatic Mapping Tutorial," in *Generative and Transformation Techniques in Software Engineering*, Braga, Portugal, 2006.

[16] C. Pablo*, et al.*, "ADO.NET entity framework: raising the level of abstraction in data programming," in *ACM SIGMOD International Conference on Management of Data*, Beijing,China, 2007, pp. 1070-1072.

[17] Hibernate. (2010). *Hibernate*. Available: http://www.hibernate.org/

[18] R. C. William and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in *27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 97-106.

[19] A. M. Russell and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in *27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 88-96.

[20] W. Gary*, et al.*, "Static checking of dynamically generated queries in database applications," *ACM Transansactions on Software Eng. Methodology,* vol. 16, p. 14, 2007.

[21] M. Andy*, et al.*, "Impact analysis of database schema changes," in *30th International Conference on Software Engineering*, Leipzig, Germany, 2008, pp. 451-460.

[22] B. Gregory*, et al.*, "Using parse tree validation to prevent SQL injection attacks," in *5th International Workshop on Software Engineering and Middleware*, Lisbon, Portugal, 2005.

[23] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.,* vol. 12, pp. 157-171, 1986.

[24] C. U. Smith and L. G. Williams, *Performance Solutions: a Practical Guide to Creating Responsive, Scalable Software*, 1st ed.: Addison Wesley, 2001.