Code Inspection

Jacopo Strada Luca Riva

January 5, 2016

Version 1.0

Contents

Listings

1 Classes

It is possible to find the inspected class at this path:

Location:*appserver/persistence/cmp/generator-database/src/main/java/com/sun/jdo/spi/persistence/generator/database/DDLGenerator.java*

<u>**Methods**</u>:

***Name**:generateDDL(SchemaElement schema, String dbVendorName, OutputStream createDDLSql, OutputStream dropDDLSql, OutputStream dropDDLJdbc, OutputStream createDDLJdbc, OutputStream dbStream, boolean dropAndCreateTbl)*

Start Line:127

***Name**:generateSQL(OutputStream createSql, OutputStream dropSql, OutputStream dropTxt, OutputStream createTxt, DatabaseOutputStream dbStream, List createAllTblDDL, List createIndexDDL, List alterAddConstraintsDDL, List alterDropConstraintsDDL, List dropAllTblDDL, String stmtSeparator, boolean dropAndCreateTbl)*

Start Line:193

***Name**:createCreateTableDDL(TableElement table, MappingPolicy mappingPolicy)*

Start Line:332

2 Functional Role

The analyzed class, *DDLGenerator* is a so-called *utility class* because it has only static methods. For this reason, we are going to explain the role of the only public method present.

The ”public static void” method *generateDDL* is used in order to create a DDL from a given schema and database vendor name. In fact the method requires a *SchemaElement* and a *String*, which is the vendor name, as parameters, in addition to them are required the various *OutputStreams* for the data elaborated by the function. The *dbStream* could also be null and, in this case, the method will not directly perform the operations on the database. Also a boolean is required and it must be set to TRUE in order to drop all the tables before creating new ones. Requesting this high number of output streams is a way to have multiple return values and leave to the caller the possibility of choosing if they would like to associate a stream to a file or to another component.

This function calls the ”private static” method *createCreateTableDDL* for each table contained in the schema received as an argument, with its *mappingPolicy*. *createCreateTableDDL* generates and returns a string from the name and the columns of the table, in order to accomplish a valid formatting of the string it uses the method *formatCreateTable* of the class *DDLTemplateFormatter* passing the name of the table. The last operation of *generateDDL* is to call the ”private static void” method *generateSQL*, whose task is writing DDL to output streams in order to drop or create tables in the database. The whole structure of the method basically consists in a ”try-finally” statement, where, if necessary, constraints and tables are dropped and created, using the *OutputStreams* and the lists received from *generateDDL*. Also the index of the tables are created.

The above description is the result of an analysis of the code considering the information contained in the *javadoc* provided by the original developers.

3 Issues

3.1 Naming Conventions

In this example it is noticeable that ”tbl” and ”table” are used as alternatives, it could be better to always use the same name, for clarity ”table”.

```java
            OutputStream dbStream, boolean dropAndCreateTbl)
```

Listing 1: DDLGenerator.java - Lines: 130 - 130

```java
            TableElement[] tables = schema.getTables();
```

Listing 2: DDLGenerator.java - Lines: 145 - 145

The same consideration can be done for ”stmt” that is often used instead of ”statement”

```java
            String stmtSeparator = mappingPolicy.getStatementSeparator();
```

Listing 3: DDLGenerator.java - Lines: 165 - 165

In the analyzed code, one-character variables are used temporarily only for loops, respecting the conventions.

In this example it is possible to notice a problem with the naming convention: the acronym ”Structured Query Language” is sometimes indicated as ”SQL” and sometimes as ”Sql”, while ”Data Definition Language” is always indicated as ”DDL”. Even though this is not a specific violation of the java naming convention it should be better to use only one way. Using ”Ddl” and ”Sql” could probably be the best solution as also in the following example it would be possible to emphasize the separation of the two acronyms: ”createDdlSql” instead of ”createDDLSql”.

```java
            OutputStream createDDLSql, OutputStream dropDDLSql,
```

Listing 4: DDLGenerator.java - Lines: 128 - 128

Constants are correctly declared using capitalized words separated by an underscore.

```java
    /** For writing DDL. */
    private static final char SPACE = ' '; //NOI18N

    /** For writing DDL. */
    private static final char START = '('; //NOI18N

    /** For writing DDL. */
    private static final char END = ')'; //NOI18N

    /** For writing DDL. */
    private static final String COLUMN_SEPARATOR = ", "; //NOI18N

    /** For writing DDL. */
    private static final String NOT_NULL_STRING = "NOT NULL"; //NOI18N

    /** For writing DDL. */
    private static final String NULL_STRING = "NULL"; //NOI18N
```

Listing 5: DDLGenerator.java - Lines: 67 - 83

3.2 Indention

In the code are usually used 4 spaces for indentation but sometimes also tabs are used as you can see (notice the arrow) in the following example:

```
⟶for (int i = 0; i < size; i++) {
```

Listing 6: DDLGenerator.java - Lines: 346 - 346

3.3 Braces

In the code is used the “Kernighan and Ritchie” style for bracing, also for one statement only conditions. Example:

```
            if (txtStream != null) {
                txtStream.close();
            }
```

Listing 7: DDLGenerator.java - Lines: 226 - 228

3.4 File Organization

The sections are correctly separated with spaces and the methods have also comments above as separators. Example:

```
    private static String createDropTableDDL(TableElement table) {
        String[] oneParam = { table.getName().getName() };
        return DDLTemplateFormatter.formatDropTable(oneParam);
    }

    /**
     * Returns DDL in String form to create a primary key constraint.  The
     * string has the format:
     * <pre>
     * CONSTRAINT pk_name PRIMARY KEY(id, name)
     * </pre>
     * @param table Table for which constraint DDL is returned.
     * @return DDL to create a PK constraint or null if there is no PK.
     */
    private static String createPrimaryKeyConstraint(TableElement table) {
        String rc = null;
        UniqueKeyElement pk = table.getPrimaryKey();

        if (pk != null) {
            String[] twoParams = new String[2];
            twoParams[0] = pk.getName().getName();
            twoParams[1] = getColumnNames(pk.getColumns());
            rc = DDLTemplateFormatter.formatPKConstraint(twoParams);
        }
        return rc;
    }
```

Listing 8: DDLGenerator.java - Lines: 390 - 415

Sometimes the line length exceeds 80 characters also if it is not needed. Some examples are reported below (line 139 and 195 go to the next line in the document because are too long):

```
139         // Added for Symfoware support as Symfoware does not automatically
    create
140         // indexes for primary keys. Creating indexes is mandatory.
```

Listing 9: DDLGenerator.java - Lines: 139 - 140

```
193 private static void generateSQL(OutputStream createSql,
194         OutputStream dropSql, OutputStream dropTxt, OutputStream createTxt,
195         DatabaseOutputStream dbStream, List createAllTblDDL, List
    createIndexDDL,
196         List alterAddConstraintsDDL, List alterDropConstraintsDDL,
197         List dropAllTblDDL, String stmtSeparator, boolean dropAndCreateTbl)
198         throws DBException, SQLException {
```

Listing 10: DDLGenerator.java - Lines: 193 - 198

3.5 Wrapping Lines

Lines are correctly wrapped by the Oracle conventions.

3.6 Comments

The *javadoc* if always present but often parameters are missing (e.g. ”createSql” at line 193) or present with a wrong name (e.g. ”dropDDLSql” insted of ”dropSQL” at lines 176 and 194). In the following example are also present comments (lines 190-192) that state the errors present in the javadoc as in the other comments.

```
173 /**
174  * Write DDL to files or drop or create table in database
175  * @param createDDLSql a file for writing create DDL
176  * @param dropDDLSql a file for writing drop DDL
177  * @param dropDDLTxt a file for writing drop DDL and can be easily
178  * executes drop in undeployment time
179  * @param dbStream for creating table in database
180  * @param createAllTblDDL a list of create table statement
181  * @param createIndexDDL a list of create index statement
182  * @param alterAddConstraintsDDL a list of adding constraints statement
183  * @param alterDropConstraintDDL a list of droping constrains statement
184  * @param dropAllTblDDL a list of droping tables statement
185  * @param stmtSeparator for separating each statement
186  * @param dropAndCreateTbl true for dropping tables first
187  * @throws DBException
188  * @throws SQLException
189  */
190 // XXX FIXME The above javadoc is wrong, change it if/when the
191 // generateDDL api changes.
192 // XXX Fix method body comments.
193 private static void generateSQL(OutputStream createSql,
194         OutputStream dropSql, OutputStream dropTxt, OutputStream createTxt,
195         DatabaseOutputStream dbStream, List createAllTblDDL, List
    createIndexDDL,
196         List alterAddConstraintsDDL, List alterDropConstraintsDDL,
197         List dropAllTblDDL, String stmtSeparator, boolean dropAndCreateTbl)
198         throws DBException, SQLException {
```

Listing 11: DDLGenerator.java - Lines: 173 - 198

Commented out code is not present in the analyzed methods.

3.7 Java Source Files

The analyzed file contains exactly one class and it is obviously the first one.

Analyzing the code it is possible to confirm that the public method of this class is implemented consistently with what is described in the javadoc.

The javadoc is also present for all the private methods in the analyzed part but, as stated already before, sometimes it is wrong.

3.8 Package and Import Statements

The package statement is the first of the file present at line 47 because before some javadoc is present. In the following lines there are import statements placed correctly.

```
47 package com.sun.jdo.spi.persistence.generator.database;
48
49 import java.io.*;
50 import java.util.*;
51 import java.sql.*;
52
53 import org.netbeans.modules.dbschema.*;
```

Listing 12: DDLGenerator.java - Lines: 47 - 53

3.9 Class and Interface Declarations

In the following lines it is summarised the declaration order present in this file.

Class documentation comment:

```
60 /**
61  * This class generates DDL for a given SchemaElement.
62  *
63  * @author Jie Leng, Dave Bristor
64  */
```

Listing 13: DDLGenerator.java - Lines: 60 - 64

Class statement:

```
65 public class DDLGenerator {
```

Listing 14: DDLGenerator.java - Lines: 65 - 65

Class static variables (only ”private static final” are present):

```
68     private static final char SPACE = ’ ’; //NOI18N
```

Listing 15: DDLGenerator.java - Lines: 68 - 68

No instance variables are present.

The constructor is not present, it would be better to add a private constructor in order to ensure the impossibility to instantiate the class.

Methods:

```
127     public static void generateDDL(SchemaElement schema, String dbVendorName,
128             OutputStream createDDLSql, OutputStream dropDDLSql,
129             OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
130             OutputStream dbStream, boolean dropAndCreateTbl)
131             throws DBException, SQLException, IOException {
```

Listing 16: DDLGenerator.java - Lines: 127 - 131

```
127     public static void generateDDL(SchemaElement schema, String dbVendorName,
128             OutputStream createDDLSql, OutputStream dropDDLSql,
129             OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
130             OutputStream dbStream, boolean dropAndCreateTbl)
131             throws DBException, SQLException, IOException {
```

Listing 17: DDLGenerator.java - Lines: 127 - 131

...

```
332     private static String[] createCreateTableDDL(TableElement table,
333             MappingPolicy mappingPolicy) {
```

Listing 18: DDLGenerator.java - Lines: 332 - 333

...

As you can see from the extracts of code present above, all the declarations of classes, methods and variables are present in a java compliant order.

For this class does not make much sense to speak about methods grouping because the class consist in only one *public static* method and all the other methods, which are private, are obviously used by the first one. For this reason all the methods present in this class are strictly related between each other.

Big duplicates are not really present, but there is a little piece of code that may be reduced creating a new method.

```
209             for (int i = 0; i < alterDropConstraintsDDL.size(); i++) {
210                 String dropConstStmt = (String) alterDropConstraintsDDL.get(i);
211                 writeDDL(workStream, txtStream, stmtSeparator, dropConstStmt);
212                 if ((dbStream != null) && dropAndCreateTbl) {
213                     dbStream.write(dropConstStmt);
214                 }
215             }
```

Listing 19: DDLGenerator.java - Lines: 209 - 215

```
218             for (int i = 0; i < dropAllTblDDL.size(); i++) {
219                 String dropTblStmt = (String) dropAllTblDDL.get(i);
220                 writeDDL(workStream, txtStream, stmtSeparator, dropTblStmt);
221                 if ((dbStream != null) && dropAndCreateTbl) {
222                     dbStream.write(dropTblStmt);
223                 }
224             }
```

Listing 20: DDLGenerator.java - Lines: 218 - 224

As you can see in the two extracts above, the code is different only because of the looped list (alterDropConstraintsDDL vs dropAllTblDDL). Resolving this problem is trivial: the only thing to do is to create a method containing the duplicated code in order to write it only once.

The longest method is *generateSql* that is 96 lines long. It is quite long but analyzing it is clear that it is not complex at all and simple to understand.

The class containing the methods assigned to us is 567 lines long. This is not a very small number but looking at the class is clear that it is really doing only one thing and it is the most important feature of a class. Being all the methods static it is very easy to split it in more classes but doing this will only bring a lack of readability.

It makes no sense to speak about *breaking encapsulation* for this class because it has no attributes that may be exposed.

The class is completely independent because it uses only basic classes of java and classes in the same package so it is not present the problem of coupling.

Cohesion is good because the class has a specific single task and all its methods are used for its unique scope.

3.10 Initialization and Declarations

Variables are of the correct type and generally referenced to their object after the declaration. In two cases there is a reference to ”null”, but the value of those variables depends on later computation, which is also managed by a ”try-finally” structure. The constructors are properly called.

```
        PrintStream workStream = null;
        PrintStream txtStream = null;

        try {
            // drop constraints first
            workStream = new PrintStream(dropSql);
            if (dropTxt != null) {
                txtStream = new PrintStream(dropTxt);
            }
```

Listing 21: DDLGenerator.java - Lines: 200 - 208

```
        } finally {
            if (workStream != null) {
                workStream.close();
            }
            if (txtStream != null) {
                txtStream.close();
            }
        }
```

Listing 22: DDLGenerator.java - Lines: 280 - 287

In this method it is possible to notice that not all the variables are declared at the beginning of the block and that there is a method’s call between the two groups of declarations.

```
    private static String[] createCreateTableDDL(TableElement table,
            MappingPolicy mappingPolicy) {

        List createTblList = new ArrayList();
        String[] oneParam = { table.getName().getName() };

        createTblList.add(
                DDLTemplateFormatter.formatCreateTable(oneParam));

⟶// add columns for each table
        ColumnElement[] columns = table.getColumns();
        String constraint = createPrimaryKeyConstraint(table);
        int size = columns.length;
```

Listing 23: DDLGenerator.java - Lines: 332 - 344

There are a few list declarations without the use of generics which were introduced in Java 5 (2004), but this class was created in 2003.

```
138            List createAllTblDDL = new ArrayList();
```

Listing 24: DDLGenerator.java - Lines: 138 - 138

3.11 Method Calls

For checking that methods are called with the correct parameters in the correct order we have to compare the call with the signature. Problems can exist only when there are two parameters of the same type because otherwise an error at compile time is suddenly presented.
Here is an example in which errors cannot be present:

```
152                            createCreateTableDDL(table, mappingPolicy));
```

Listing 25: DDLGenerator.java - Lines: 152 - 152

```
332    private static String[] createCreateTableDDL(TableElement table,
333            MappingPolicy mappingPolicy) {
```

Listing 26: DDLGenerator.java - Lines: 332 - 333

Instead below you can see a method that may cause problems if not called properly because there are two parameters of the same type, no errors will be presented neither at compile time nor at run time but it will produce wrong files:

```
298    private static void writeDDL(PrintStream sql, PrintStream txt,
299            String stmtSeparator, String stmt) {
```

Listing 27: DDLGenerator.java - Lines: 298 - 299

In the analyzed code all the methods are called correctly.

Checking the code we have not found methods called wrongly due to the presence of another one with a similar name.

The only method with a return value is the following one:

```
332    private static String[] createCreateTableDDL(TableElement table,
333            MappingPolicy mappingPolicy) {
```

Listing 28: DDLGenerator.java - Lines: 332 - 333

The return type is an array of Strings and the method is used only once. That time the return value is simply added in a list. So, no misuses of return values are present.

3.12 Arrays

Checking the code it is possible to notice that all the arrays are correctly used. In fact, as you can see from the code below, the loop is initialized from zero in order to respect the array indexing.

```java
            TableElement[] tables = schema.getTables();

            if (tables != null) {
                for (int ii = 0; ii < tables.length; ii++) {
                    TableElement table = tables[ii];
```

Listing 29: DDLGenerator.java - Lines: 145 - 149

3.13 Object Comparison

All the objects are correctly compared using the *equals* method instead of ==.

3.14 Output Format

This piece of code doesn't generate a *System out* output but generates a *Stream.* The only words present as *Strings*, which are responsible of errors, are declared as global variables and it is very easy to check that they are correct. Spaces and commas are correctly interposed.

Errors are never handled and so only well known java exceptions are thrown to the caller without adding details about the error.

3.15 Computation, Comparisons and Assignments

In these functions there is not a vast use of arithmetic operators, the used ones are "!=" and "¡" respectively for comparisons with "Null" and cycles' termination management. The operators are always used correctly. There are no try-catch structures but only try-finally.

3.16 Exceptions

The relevant exceptions present in the analyzed code are: *DBException*, *SQLException* and *IOException.*

All of them are not caught but only thrown to the caller; only sometimes a finally statement is present in order to always execute some necessary commands for closing streams.

3.17 Flow of Control

There are no switch case constructs in the analyzed class.

There are only *for* loops in the analyzed code, so, for their nature, it is very easy to verify initializations, increments and exit conditions. All the loops are formed correctly.

3.18 Files

There are no files present.

4 Other Problems

Analyzing the code we have found some other extracts which are not wrong but their construction could make some problems difficult to debug.

The first problem occurs when a null *schema* is passed in the *generateDDL* method. In this case the method do nothing and return immediately.

```
127     public static void generateDDL(SchemaElement schema, String dbVendorName,
128             OutputStream createDDLSql, OutputStream dropDDLSql,
129             OutputStream dropDDLJdbc, OutputStream createDDLJdbc,
130             OutputStream dbStream, boolean dropAndCreateTbl)
131             throws DBException, SQLException, IOException {
132 
133         if (schema != null) {
```

Listing 30: DDLGenerator.java - Lines: 127 - 133

...

```
170         }
171     }
```

Listing 31: DDLGenerator.java - Lines: 170 - 171

It would be better if an else statement was added in order to throw an exception specifying the problem.

Another problem like the one just described is the behavior of the same method when a null *dbVendorName* or a vendor name not supported is passed. In the null case the default one (SQL92) is used, instead, if the vendor name is not supported an *IOException* is thrown. This things are not specified in the javadoc as well as the list of supported vendors and the exception is not clear at all.